Optimización de rendimiento, justicia y consumo energético en sistemas multicore asimétricos mediante planificación.

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas.

Autor

Adrián Pousa

Directores

Ing. Armando E. De Giusti [1]

Dr. Juan Carlos Sáez Alcaide [2]

La Plata, Junio de 2017

Facultad de Informática - Universidad Nacional de La Plata

Dirección conjunta por convenio de colaboración

Universidad Nacional de La Plata - Universidad Complutense de Madrid.

[1] III-LIDI, Facultad de Informática, Universidad Nacional de La Plata.

[2] Dacya, Facultad de Informática, Universidad Complutense de Madrid.

Este trabajo de investigación fue realizado en el marco del proyecto *"Formación en computación avanzada"* (B/024918/09) de colaboración entre la Universidad Nacional de La Plata (Argentina) y la Universidad Complutense de Madrid (España), y los proyectos acreditados por el programa de incentivos de la UNLP *"Arquitecturas multiprocesador en HPC: software de base, métricas y aplicaciones"* (11/F018) y *"Cómputo paralelo de altas prestaciones. Fundamentos y evaluación de rendimiento en HPC. Aplicaciones a sistemas inteligentes, simulación y tratamiento de imágenes"* (11/F017).

Asimismo, este trabajo ha sido financiado por la Universidad Nacional de La Plata a través de subsidios de viajes y estadías, por la Unión Europea (FEDER) y el Ministerio de Economía, Industria y Competitividad (MINECO) de España bajo los proyectos TIN2012-32180 y TIN2015-65277-R, así como por la Red Europea de Excelencia HIPEAC-4.

Agradecimientos

En primer lugar, agradezco a mis directores de tesis Ing. Armando De Giusti y Dr. Juan Carlos Sáez Alcaide, por su tiempo y dedicación a lo largo de este proceso. Agradezco la confianza que depositaron en mí a la hora de realizar este doctorado.

También doy gracias al directorio y a mis compañeros del III-LIDI (Instituto de Investigación en Informática LIDI) por el apoyo recibido durante todo el desarrollo de esta tesis doctoral.

Del mismo modo, quiero agradecer a Francisco Tirado, Manuel Prieto y, en general, al grupo de investigación ArTeCS (Group of Architecture and Technology of Computing Systems) de la UCM por lo bien que me han recibido y por brindarme esta oportunidad de colaboración.

Deseo agradecer de manera especial a toda mi familia por su apoyo incondicional a lo largo de estos años, y en especial por su paciencia y comprensión durante mis largas estadías de trabajo en España.

A todos ellos muchas gracias.

Resumen

Los procesadores multicore asimétricos o AMPs (*Asymmetric Multicore Processors*) constituyen una alternativa de bajo consumo energético a los procesadores multicore convencionales, que están formados por cores idénticos. Los AMPs integran cores rápidos y complejos de alto rendimiento, y cores más simples de bajo consumo, que ofrecen menores prestaciones.

Investigaciones previas han demostrado que los AMPs ofrecen numerosos beneficios frente a los multicores convencionales, pero también plantean importantes desafíos para el software de sistema. Gran parte de las optimizaciones del software de sistema propuestas para AMPs se han realizado a nivel de sistema operativo (SO), principalmente mediante la inclusión de algoritmos de planificación conscientes de la asimetría en la plataforma. Una ventaja de esta aproximación es el hecho de que las aplicaciones no requieren modificaciones para explotar de forma efectiva los beneficios de los AMPs.

La mayor parte de los algoritmos de planificación existentes para AMPs intentan optimizar el rendimiento global. Sin embargo, estos algoritmos degradan otros aspectos críticos como la justicia o la eficiencia energética de la plataforma. Además, dada su naturaleza, resulta complejo extender estas estrategias para soportar prioridades de usuario. El principal objetivo de esta tesis doctoral es superar estas limitaciones, mediante el diseño de estrategias de planificación más flexibles para AMPs. Asimismo, en esta tesis realizamos diversos estudios que muestran el impacto que la optimización de una métrica tiene en otras.

Para mejorar el rendimiento global, la justicia o la eficiencia energética en AMPs, el planificador debe tener en cuenta el beneficio que cada aplicación alcanza al usar los distintos tipos de cores en un AMP. Se ha demostrado que no todos los hilos en ejecución de una carga de trabajo obtienen siempre el mismo beneficio relativo (*speedup factor – SF*) al usar un core de alto rendimiento frente a uno de bajo consumo. Tener en cuenta esta diversidad de *SFs* a la hora de distribuir los cores entre aplicaciones es clave para optimizar los distintos objetivos. Para esto, el SO debe determinar de forma efectiva el *SF* de cada hilo en tiempo de ejecución. Para hacer frente a este desafío, en esta tesis proponemos una metodología general para construir modelos de estimación de *SF* precisos basados en el uso de contadores hardware.

La mayoría de los algoritmos de planificación existentes para AMPs, han sido evaluados empleando simuladores o plataformas asimétricas emuladas. Muchas de estas estrategias se han evaluado utilizando prototipos de planificadores en modo usuario. Por el contrario, en esta tesis doctoral, evaluamos los algoritmos propuestos en un entorno más realista: empleando implementaciones de los algoritmos en el kernel de SOs reales (OpenSolaris y GNU/Linux) y sobre hardware multicore asimétrico real.

Lista de publicaciones

- J. C. Saez, M. Prieto-Matías, A. Pousa, and A. Fedorova, "Explotación de técnicas de especialización de cores para planificación eficiente en procesadores multicore asimétricos", *Jornadas de paralelismo*, 2011.
- A. Pousa, J. C. Saez, A. D. Giusti, and M. Prieto-Matías, "Evaluation of scheduling algorithms on an asymmetric multicore prototype system", *XX Congreso Argentino de Ciencias de la Computación*, 2014.
- J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto Matías, "Exploring the throughput-fairness trade-off on asymmetric multicore systems", *Proc. of Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Part II*, 2014.
- J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matías, "ACFS: A completely fair scheduler for asymmetric single-ISA multicore systems", *Proceedings of ACM/SIGAPP Symposium On Applied Computing*, 2015.
- J. C. Saez, A. Pousa, R. Rodriguez-Rodriguez, F. Castro, and M. Prieto-Matías, "PMCtrack: Delivering performance monitoring counter support to the OS scheduler", *The Computer Journal*, 2016.
- A. Pousa, J. C. Saez, F. Castro, D. Chaver, and M. Prieto-Matías, "Towards completely fair scheduling on asymmetric single-ISA multicore processors", *Journal of Parallel and Distributed Computing*, 2017.
- J. C. Saez, A. Pousa, A. D. Giusti, and M. Prieto-Matías, "On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors", *The Computer Journal*, 2017.

Índice general

1 Introducción

La mayoría de los procesadores de propósito general actuales están compuestos por varios núcleos (o *cores*) que residen en un mismo circuito integrado. Estos procesadores se conocen como procesadores multicore o CMPs (*Chip Multi-Processors*). La mayoría de los CMPs comerciales son simétricos, es decir, están compuestos por cores idénticos, y pueden dividirse en dos grandes grupos: aquellos formados por cores complejos desde el punto de vista de la microarquitectura (como los de la familia Haswell de Intel o Power8 de IBM), y aquellos formados por cores más simples de consumo reducido (como ARM Cortex A9 o Intel Xeon Phi). Los procesadores del primer grupo implementan características microarquitectónicas sofisticadas, como ejecución fuera de orden y superescalar, que permiten incrementar el rendimiento de un único hilo de forma significativa. Lamentablemente, estudios previos muestran que los procesadores de este tipo no podrán integrar más allá de un número reducido de cores debido a problemas de consumo y disipación [30]. Los procesadores del segundo grupo poseen mejores prestaciones para aplicaciones con un elevado paralelismo a nivel de hilo (TLP), pero ofrecen un rendimiento inferior para aplicaciones secuenciales [23].

En la actualidad los CMPs se encuentran en un amplio espectro de plataformas, desde dispositivos móviles hasta *datacenters*. En estas plataformas se ejecutan cargas de trabajo muy diversas, como procesamiento multimedia, encriptación, aplicaciones de cálculo científico, servidores web y aplicaciones de *cloud computing* [40]. Los distintos tipos de cargas de trabajo presentan distintas demandas y perfiles de cómputo, y exigen al software de sistema que optimice distintos objetivos, a veces contrapuestos, como el rendimiento global, el tiempo de respuesta, la justicia o la eficiencia energética.

Teniendo en cuenta esta gran diversidad de cargas de trabajo, un modelo específico de procesador multicore simétrico puede resultar ideal para un conjunto específico de aplicaciones pero no para todas. En particular, los cores rápidos y complejos son adecuados para aplicaciones intensivas en cómputo que pueden obtener beneficios significativos de la microarquitectura sofisticada que poseen estos cores. Estas aplicaciones utilizan eficiente-

mente el pipeline del procesador, tienen un alto grado de paralelismo a nivel de instrucción y generalmente tienen una buena localidad en el acceso a memoria. Por otro lado, los cores lentos, simples y de consumo reducido pueden resultar más adecuados para aplicaciones intensivas en memoria en términos de rendimiento por *watt*. En general, estas aplicaciones utilizan los pipelines complejos de manera ineficiente, ocasionando frecuentes paradas del pipeline debido a numerosos fallos de cache. Como es evidente, la elección del tipo de sistema multicore *simétrico*, constituido por unos pocos cores complejos o bien por abundantes cores simples de bajo consumo, determina el tipo de aplicación para el que el diseño elegido proporciona un mayor rendimiento por *watt* [29, 23].

Los procesadores multicore asimétricos o AMPs (*Asymmetric Multicore Processors*) han sido propuestos como alternativa a los CMPs convencionales para ofrecer un mejor rendimiento por *watt* y por área [29], así como para satisfacer las distintas demandas de cargas de trabajo diversas [30, 8]. Un AMP típico está compuesto por dos tipos de core: cores rápidos de alto rendimiento (*big o fast cores*) y cores lentos de bajo consumo (*small o slow cores*). Para simplificar el desarrollo de software, todos los cores de un AMP poseen el mismo repertorio de instrucciones (ISA - *Instruction-Set Architecture*).[1] Por este motivo algunos autores [41, 31, 11, 40] hacen también referencia a los AMPs mediante los términos "Asymmetric Single-ISA multicores" o "ASISA". Cabe destacar que, como alternativa a los AMPs, existen otras arquitecturas heterogéneas comerciales, como el procesador Cell/BE de IBM [19], donde los distintos cores exponen un repertorio de instrucciones disjunto (*heterogenous-ISA* CMP). La existencia de cores con distinto repertorio de instrucciones introduce desafíos significativos para el programador [43]. Este tipo de arquitecturas heterogéneas está fuera del alcance de esta tesis doctoral, que se centra sólo en los AMPs.

Varios trabajos han mostrado los beneficios que ofrecen los AMPs con respecto a los CMPs simétricos [30, 3, 23, 41, 64, 20]. Un resumen de estos puede encontrarse en [40]. A continuación, describimos algunos de los beneficios que los AMPs ofrecen en el contexto del cómputo de altas prestaciones, que es el principal enfoque de esta tesis doctoral.

Como se mencionó anteriormente, los CMPs pueden proporcionar buen rendimiento por *watt* para algunas aplicaciones pero no para todas. En un AMP, es posible superar esta limitación mediante técnicas de *especialización* [58, 53, 20]; es decir garantizando que una aplicación se ejecute en el tipo de core que ofrece la mejor relación entre rendimiento y consumo energético. Para esto es necesario tener en cuenta las características de cada programa [29, 30, 8, 60].

En general, en un AMP los cores simples son adecuados para la ejecución de aplicaciones paralelas altamente escalables. Por el contrario, los cores complejos son adecuados para la ejecución de aplicaciones secuenciales, ya que éstas no pueden mejorar su rendimiento distribuyendo el cómputo a través de múltiples cores simples. En el caso de aplicaciones

[1] Utilizar el mismo ISA permite ejecutar el mismo binario en los distintos cores sin tener que recompilar el código para cada uno de ellos.

paralelas que tienen fases secuenciales, es posible utilizar los cores rápidos para acelerar estas fases [3, 54]. Debido a la relación rendimiento/consumo entre los cores rápidos y los cores lentos, resulta mucho más eficiente ejecutar una aplicación paralela en un gran número de cores simples que en un número reducido de cores complejos que ocupen la misma área y poseen el mismo consumo de potencia agregado [23, 58].

En el resto de este capítulo introductorio presentamos la motivación, los objetivos y la organización de esta tesis doctoral.

1.1. Motivación

A pesar de sus beneficios, los AMPs plantean importantes desafíos para el software de sistema [17, 50, 10, 40]. Uno de los principales desafíos es cómo distribuir eficientemente los ciclos de los cores rápidos y lentos entre las distintas aplicaciones que se ejecutan en el sistema. Esta responsabilidad recae principalmente en el planificador del sistema operativo.

Al inicio de esta tesis doctoral, la mayoría de los algoritmos de planificación propuestos para AMPs tenían como objetivo optimizar el rendimiento global [30, 8, 54, 28, 58, 59]. Para alcanzar este objetivo, el planificador debe ejecutar en los cores rápidos aquellas aplicaciones que usan estos cores de forma eficiente, ya que estas aplicaciones obtienen mejoras significativas de rendimiento (*speedup*) con respecto a cuando se ejecutan en los cores lentos [30]. Por otra parte, es posible obtener mejoras adicionales usando los cores rápidos para acelerar las fases secuenciales existentes en aplicaciones paralelas [3, 54].

En esta tesis doctoral demostramos que los planificadores que intentan optimizar únicamente el rendimiento global, son inherentemente injustos. La injusticia genera varios efectos no deseados en el sistema [16, 42, 13]. Por ejemplo, al utilizar un algoritmo de planificación injusto, una aplicación puede experimentar grandes variaciones de rendimiento entre distintas ejecuciones en un AMP, ya que depende enormemente de la naturaleza del resto de las aplicaciones que se ejecutan con ella. Asimismo, aplicaciones con la misma prioridad pueden no experimentar la misma degradación en rendimiento (*slowdown*) en la carga de trabajo, con respecto a cuando se ejecutan solas en el AMP. Este hecho provoca que las estrategias de gestión de prioridades resulten completamente ineficaces [13], y podrían llevar a facturaciones erróneas en servicios de cómputo sobre *Cloud*, donde se factura a los usuarios por tiempo de uso de CPU. Estos problemas pueden mitigarse en AMPs con políticas de planificación conscientes de la justicia (*fairness-aware schedulers*).

Investigaciones recientes han demostrado que optimizar el rendimiento global en un AMP no siempre garantiza un buen compromiso entre rendimiento y consumo de energía [20, 65]. Específicamente, usar los cores rápidos en un AMP para ejecutar las aplicaciones de la carga de trabajo que alcanzan el mayor beneficio relativo (*speedup*) con respecto a cores lentos, no siempre constituye la mejor estrategia desde el punto de vista de la eficiencia energética.

1.2. Objetivos y principales desafíos

Aunque el problema de maximizar el rendimiento de distintas cargas de trabajo en AMPs ha sido ampliamente estudiado, el análisis de otros aspectos críticos del sistema como la optimización de la justicia o de la eficiencia energética, así como el soporte efectivo de prioridades no ha recibido suficiente atención por parte de la comunidad científica. El principal objetivo de esta tesis doctoral es analizar estos aspectos en detalle, y diseñar estrategias de planificación a nivel de sistema operativo para optimizar la justicia y la eficiencia energética en un AMP. Asimismo, pretendemos que estos algoritmos de planificación trasladen los beneficios de los AMPs directamente a las aplicaciones, sin requerir conocimiento previo de estas, ni modificación de las mismas, ni extensiones especiales del hardware.

Para lograr estos objetivos fue necesario superar tres grandes desafíos. En primer lugar, desarrollamos un *framework* de planificación complejo para facilitar la implementación y evaluación de las distintas estrategias de planificación en un entorno lo más realista posible: empleando un sistema operativo (SO) de propósito general sobre hardware multicore asimétrico real. El desarrollo de este *framework* demandó una gran cantidad de tiempo durante el transcurso de esta tesis doctoral. En general, realizar modificaciones profundas en un componente central del kernel de un sistema operativo, como es el planificador, resulta mucho más complejo que el desarrollo tradicional de aplicaciones en espacio de usuario. Para evitar hacer frente a esta complejidad, muchos investigadores han evaluado estrategias de planificación para AMPs utilizando simuladores [30, 8, 11, 65] o prototipos de planificación en espacio de usuario [45, 46]. Investigaciones previas en el área han mostrado que las implementaciones en sistemas reales permiten detectar problemas significativos en algoritmos que han sido evaluados previamente mediante simuladores.[2] Durante esta tesis doctoral, emplear un SO real para nuestro estudio ha permitido identificar limitaciones significativas en algunas propuestas previas [11]. Asimismo, la gran flexibilidad del framework de planificación ha hecho posible realizar análisis experimentales profundos sobre hardware asimétrico real, que no estaba disponible al inicio de la tesis doctoral, como es el caso del prototipo QuickIA [10] de Intel o el de diversas placas de desarrollo que integran procesadores big.LITTLE[5] de ARM.

El segundo desafío consiste en proveer al planificador del SO de un mecanismo para estimar en tiempo de ejecución el beneficio relativo (*speedup*) que una aplicación obtiene al usar los distintos tipos de cores en un AMP. En una carga de trabajo multiprogramada, cada una de las aplicaciones puede obtener beneficios diferentes al ejecutarse en cores rápidos con respecto a hacerlo en los cores lentos [8, 58]. Para ilustrar este hecho, la figura 1.1 muestra el ratio de rendimiento observado al ejecutar las distintas aplicaciones de la suite SPEC CPU2006 en cores rápidos y lentos del prototipo QuickIA de Intel. Los detalles de esta plataforma pueden encontrarse en el capítulo 2. Como puede observarse en la figura, hay aplicaciones que no utilizan de forma efectiva los cores rápidos y complejos (como `lbm` o `libquantum`) lo que se traduce en un beneficio relativo muy reducido. Otras aplicaciones, como `calculix` o

[2] Este es el caso, por ejemplo, de las conclusiones obtenidas por Shelepov y otros [60, 59] al implementar el algoritmo IPC-driven [8] en un sistema real.

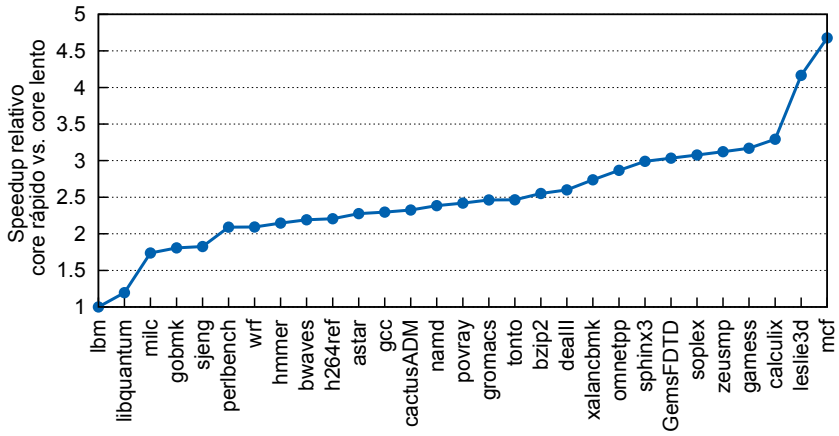


Figura 1.1: Beneficio relativo (core rápido vs. core lento) asociado a cada una de las aplicaciones de la suite SPEC CPU2006 en el prototipo QuickIA de Intel.

`mcf`, experimentan un gran beneficio al ejecutarse en un core rápido frente a uno lento, esto se debe principalmente a que explotan eficientemente los recursos de hardware adicionales presentes en los cores de alto rendimiento. Dada la gran diversidad de *speedups* que puede estar presente en una carga de trabajo multiprogramada, realizar asignaciones de aplicaciones a cores en base al *speedup* de cada aplicación es clave para mejorar el rendimiento global, la justicia o la eficiencia energética en AMPs [53, 57, 46].

En una aplicación puramente secuencial, como los benchmarks de SPEC CPU2006, el beneficio relativo (*speedup*) es simplemente el ratio del número de instrucciones por segundo (IPS) que el único hilo en ejecución de la aplicación experimenta en ambos tipos de core: $\frac{IPS_{rápido}}{IPS_{lento}}$. Este ratio recibe el nombre de factor de ganancia o *SF* (*Speedup Factor*) [58] del hilo de ejecución. En trabajos previos [60, 59] se ha demostrado que medir directamente el *SF*, mediante la monitorización del rendimiento del hilo en ambos tipos de core, da lugar a imprecisiones en el cálculo del *SF* y genera un *overhead* significativo en escenarios multi-aplicación. Para hacer frente a este problema en esta tesis, optamos por construir modelos de estimación de *SF* basados en la monitorización del rendimiento del hilo solamente en el core donde se encuentra ejecutándose actualmente. Como mostramos en el capítulo 6, la construcción de estos modelos de estimación constituye un gran desafío, especialmente en AMPs donde los distintos cores poseen diferencias profundas a nivel microarquitectónico. Otra dificultad asociada es la aproximación del beneficio relativo (*speedup*) que una aplicación multi-hilo como un todo experimenta al usar todos los cores de la plataforma con respecto a usar cores lentos únicamente. En esta tesis, construimos modelos analíticos para aproximar el *speedup* de distintos tipos de aplicaciones paralelas.

El tercer gran desafío de esta tesis ha sido la selección de métricas adecuadas para cuantificar el grado de alcance de los distintos objetivos del planificador (rendimiento global, justicia y eficiencia energética) en un AMP. La mayor parte de las métricas existentes para capturar la efectividad de las estrategias de planificación en distintos escenarios, han sido

definidas específicamente para sistemas simétricos, pero no son aptas para AMPs. Por este motivo, para llevar a cabo nuestros estudios analíticos y experimentales durante la tesis, fue necesario construir nuevas métricas específicas para AMPs, y adaptar métricas usadas en sistemas simétricos a entornos AMP. Empleando estas métricas, derivamos modelos analíticos y herramientas auxiliares para aproximar la planificación teórica que optimiza distintos objetivos. Esto resultó esencial para guiar el proceso de diseño de algunas de las estrategias de planificación propuestas en esta tesis, como es el caso de los algoritmos ACFS y EEF-Driven.

1.3. Contribuciones de la tesis

Las principales contribuciones de la tesis doctoral son las siguientes:

- Proponemos los algoritmos de planificación para AMPs Prop-SP, ACFS y EEF-Driven, que persiguen distintos objetivos. El algoritmo Prop-SP constituye nuestra primera propuesta de planificación orientada a justicia que tiene en cuenta la diversidad de *speedups* entre aplicaciones para (1) ofrecer un buen equilibrio entre justicia y rendimiento global, y (2) proporcionar soporte a prioridades [52, 57]. Aunque Prop-SP ofrece mejores resultados que otras estrategias de planificación propuestas previamente, este algoritmo no es capaz de *optimizar* la justicia en la plataforma. Esto se debe en parte a que la optimización de la justicia y del rendimiento son objetivos contrapuestos en AMPs, como demostramos en esta tesis. Para superar esta limitación creamos el planificador ACFS, que está diseñado para optimizar la justicia en un AMP. Este planificador permite además ajustar gradualmente el nivel relativo de justicia y rendimiento global obtenido al ejecutar una carga de trabajo multiprogramada en un AMP. Finalmente, diseñamos el algoritmo de planificación EEF-Driven, que persigue optimizar la eficiencia energética en AMPs.

- Además de las estrategias mencionadas anteriormente proponemos el planificador ACFS-E, que permite la optimización de tres aspectos independientes (justicia, rendimiento global y eficiencia energética) con un único algoritmo de planificación. Este planificador (variante de ACFS) está equipado con un conjunto de parámetros configurables que permiten al administrador del sistema obtener mayor eficiencia energética o rendimiento global a costa de degradar la justicia de forma gradual en el AMP.

- En esta tesis construimos también un modelo analítico para aproximar el rendimiento global, el grado de justicia y la eficiencia energética que una estrategia de planificación puede ofrecer cuando se utilizan cargas de trabajo multiprogramadas en AMPs. Empleando este modelo analítico y un simulador basado en él realizamos diversos estudios para hallar el planificador teórico que optimiza cada una de las tres métricas. Este estudio ha sido clave para guiar el proceso de diseño de la mayor parte de los algoritmos propuestos en esta tesis, que intentan aproximar distintos planificadores óptimos. Asimismo, el estudio muestra la interrelación entre el rendimiento global, el grado de

justicia y la eficiencia energética en AMPs, y nos permite concluir que optimizar dos o más de estas métricas simultáneamente no es posible en la mayoría de los casos.

- Por otra parte, proponemos una metodología para guiar el proceso de construcción de modelos precisos de estimación de *SF* basados el uso de contadores hardware. Gran parte de los algoritmos de planificación evaluados en esta tesis, realizan asignaciones de hilos a los distintos tipos de cores basándose en estos modelos de estimación. Hemos observado que otras metodologías propuestas previamente para construir estos modelos de estimación [53] no resultan efectivas en AMPs donde los distintos cores presentan diferencias muy profundas a nivel microarquitectónico y en la jerarquía cache, como es el caso del prototipo QuickIA de Intel. La metodología propuesta en esta tesis permite construir modelos precisos, incluso en escenarios complejos. Esta metodología resulta también efectiva para construir modelos de estimación del *factor de eficiencia energética* o *EEF* (*Energy-Efficiency Factor*) de un hilo, que introducimos en el capítulo 7. Los algoritmos de planificación ACFS-E y EEF-Driven necesitan aproximar este factor en tiempo de ejecución.

- Una de las principales diferencias de los algoritmos Prop-SP y ACFS con respecto a otros algoritmos orientados a justicia [33, 64] es que nuestras propuestas ofrecen soporte específico para acelerar distintos tipos de aplicaciones paralelas. Incorporar este soporte en el planificador ha constituido un importante desafío de diseño, que ha requerido considerar dos aspectos fundamentales. En primer lugar, observamos que no todas las aplicaciones multi-hilo obtienen beneficios significativos si el planificador asigna la misma fracción de tiempo de cores rápidos y lentos a todos los hilos. Por este motivo, es necesario que el planificador ofrezca distintas estrategias de distribución de ciclos de core rápido entre hilos para trasladar los beneficios de los AMPs a diversos tipos de aplicaciones paralelas. Asimismo, demostramos que la interacción entre el SO y el *runtime system* permite obtener beneficios adicionales, especialmente en aplicaciones OpenMP regulares. En segundo lugar, el *SF* de cada hilo de una aplicación paralela no aproxima el beneficio que la aplicación como un todo obtiene al usar los escasos cores rápidos en un AMP. Además, este beneficio depende del tipo de aplicación paralela y no varía de la misma forma al aumentar el número de hilos en la aplicación. En esta tesis derivamos modelos analíticos para aproximar, para distintos tipos de aplicaciones paralelas, el beneficio relativo (*speedup*) que se obtiene al usar cores rápidos. Los distintos algoritmos de planificación emplean estos modelos en tiempo de ejecución para ofrecer mejor soporte en cargas de trabajo multiprogramadas que incluyen aplicaciones multi-hilo.

- Para evaluar la efectividad de las distintas estrategias de planificación propuestas en esta tesis, implementamos los algoritmos en un SO real (usando el *framework* de planificación desarrollado) y realizamos una comparación exhaustiva con otros algoritmos[3]

[3] Muchos de estos algoritmos existentes habían sido evaluados originalmente empleando simuladores, lo cual demandó que realizáramos la implementación de los mismos en el kernel de un SO real.

del estado del arte [8, 33, 28, 58, 59, 65, 64]. Para la evaluación experimental utilizamos distintas plataformas hardware. Durante la primera etapa de la tesis, no existía hardware multicore asimétrico comercial. En esta etapa fue necesario emular sistemas multicore asimétricos mediante la reducción de la frecuencia de algunos cores ("lentos") en CMPs simétricos. Una vez que tuvimos a nuestra disposición hardware AMP real, como el prototipo Intel QuickIA o sistemas equipados con procesadores big.LITTLE de ARM, evaluamos los distintos algoritmos usando este tipo de plataformas.

1.4. Organización de la tesis

El resto de la tesis se estructura como sigue:

- El capítulo 2 presenta las características del entorno utilizado a lo largo de esta tesis para llevar a cabo nuestra evaluación experimental. Este entorno está compuesto por el hardware asimétrico, el framework de planificación para AMPs y un conjunto de herramientas auxiliares. En esta tesis utilizamos dos sistemas operativos para llevar a cabo nuestro análisis experimental: OpenSolaris y GNU/Linux. En este capítulo describimos las características del planificador de estos dos sistemas y presentamos las principales modificaciones necesarias para incorporar nuestro framework en cada SO.
- En el capítulo 3 discutimos el trabajo relacionado y describimos las métricas empleadas para cuantificar la efectividad de los distintos planificadores analizados en esta tesis.
- El capítulo 4 se centra en la descripción y análisis del algoritmo Prop-SP, el primer algoritmo orientado a justicia en AMPs que explota la diversidad de *speedups* entre aplicaciones para lograr sus objetivos. En este capítulo describimos también en detalle el soporte que Prop-SP ofrece para acelerar aplicaciones multi-hilo y presentamos las fórmulas (derivadas analíticamente) para determinar el *speedup* de una aplicación multi-hilo.
- En el capítulo 5 presentamos un modelo teórico para aproximar de forma analítica el rendimiento global y el grado de justicia que un algoritmo de planificación puede extraer de un AMP al ejecutar cargas de trabajo multiprogramadas. Empleando este modelo, realizamos un estudio analítico que pone de manifiesto que no es posible optimizar ambos aspectos simultáneamente. El estudio también revela que, aunque Prop-SP ofrece un buen compromiso rendimiento-justicia no es capaz de optimizar la justicia en un AMP.
- El capítulo 6 describe el algoritmo de planificación ACFS, diseñado para optimizar la justicia en un AMP y mejorar el soporte ofrecido por Prop-SP. En este capítulo también presentamos una nueva metodología sistemática para generar modelos de estimación del *SF* en AMPs. Esta metodología permite construir modelos de estimación precisos para aproximar el *SF* desde los distintos cores del prototipo QuickIA de Intel.

- En el capítulo 7 ampliamos el modelo teórico presentado en el capítulo 5 para aproximar el grado de eficiencia energética que ofrecen distintas estrategias de planificación. Empleando el modelo teórico extendido realizamos un estudio que muestra la interrelación entre la justicia, la eficiencia energética y el rendimiento global en AMPs. Los resultados obtenidos en este estudio permiten modelar el *factor de eficiencia energética* o *EEF* (*Energy-Efficiency Factor*) de un hilo, una métrica que el planificador del SO puede explotar de forma efectiva para mejorar la eficiencia energética que se extrae de un AMP. En este capítulo también presentamos los algoritmos EEF-Driven y ACFS-E, que utilizan el *EEF* de cada hilo para realizar asignaciones de hilos a cores y de esta forma reducir el producto energía-retardo (EDP). Finalmente, realizamos un análisis experimental exhaustivo empleando sistemas con procesadores big.LITTLE de ARM que muestra los beneficios de EEF-Driven y ACFS-E frente a estrategias de planificación previamente propuestas.
- El capítulo 8 expone las principales conclusiones de la tesis y discute el trabajo futuro.
- El Anexo A muestra el proceso de derivación de las fórmulas para estimar el *speedup* para una aplicación multi-hilo.
- El Anexo B describe en detalle el proceso de división en fases de las trazas de *SF* de una aplicación. Este proceso es parte de la metodología presentada en el capítulo 6 para derivar modelos de estimación de *SF*.

2 Entorno experimental

En este capítulo presentamos las características del entorno experimental utilizado a lo largo de esta tesis doctoral, el cual está compuesto por el hardware asimétrico y el framework de planificación para AMPs.

Al inicio de esta tesis doctoral no existían multicores asimétricos comerciales. Por esta razón, muchos investigadores y desarrolladores han recurrido a diferentes técnicas para emular o simular estas arquitecturas [30, 60, 8]. Hoy en día, existen multicores asimétricos comerciales como el procesador *big.LITTLE* de ARM [5], que se integra en múltiples plataformas móviles. También existen algunos prototipos, como por ejemplo el sistema *QuickIA* de Intel [10].

Gran parte de las optimizaciones del software de sistema propuestas para AMPs se han realizado a nivel de sistema operativo, principalmente mediante la inclusión de algoritmos de planificación conscientes de la asimetría en la plataforma [60, 54, 28]. De esta forma es posible trasladar los beneficios de los AMPs a las aplicaciones sin requerir su modificación. No obstante, esto supone un reto significativo para los diseñadores de SOs. Para dar soporte a AMPs en un sistema operativo de propósito general es necesario modificar el planificador de forma sustancial. Esto representa una tarea muy compleja, ya que conlleva trabajar a nivel de kernel. Asimismo, se requieren herramientas específicas de apoyo para el desarrollo y la depuración.

Es importante mencionar que en la actualidad la mayoría de los sistemas operativos no ofrecen soporte para AMP. En particular, una excepción muy clara es el parche `SCHED_HMP` de Linux que está destinado a proporcionar soporte para procesadores *big.LITTLE* de ARM en entornos interactivos de dispositivos móviles [49]. Sin embargo, este soporte es muy limitado porque no está pensado para cargas de trabajo intensivas en cómputo, como las que aparecen en el contexto del cómputo de altas prestaciones (HPC).

Comenzamos este capítulo describiendo el hardware asimétrico utilizado. A continuación describimos nuestro framework de planificación para multicores asimétricos (implementado

Plataforma	Procesadores	Cores por procesador	Caché de ultimo nivel
Intel-8	2 x Intel Xeon X5365 (Clovertown)	4	4MB (L2)
AMD-12	2 x AMD Opteron 2435 (Istambul)	6	6MB (L3)
Intel QuickIA	Intel Xeon E5450	4	6MB (L3)
	Intel Atom N330	2	1MB (L2)
ARM Juno	ARM Cortex A57	2	2MB (L2)
	ARM Cortex A53	4	1MB (L2)

Tabla 2.1: Características de las plataformas utilizadas

en dos sistemas operativos: OpenSolaris y GNU/Linux) y enumeramos las modificaciones realizadas en los distintos sistemas para dar soporte para AMPs. Por último, describimos las herramientas auxiliares utilizadas.

2.1. Hardware asimétrico

En la tabla 2.1 se muestran las características de las plataformas hardware utilizadas en los distintos experimentos realizados durante el transcurso de esta tesis doctoral. Como puede observarse, empleamos tanto plataformas multicore simétricas (Intel-8 y AMD-12) como asimétricas (Intel QuickIA y ARM Juno).

Al inicio de esta tesis utilizamos arquitecturas multicores simétricas debido a que no existían plataformas asimétricas comerciales. Para emular la asimetría en un sistema simétrico reducimos la frecuencia de un subconjunto de cores, para obtener "cores lentos" en el sistema. Durante el desarrollo de la tesis surgieron algunas plataformas asimétricas reales con las cuales pudimos experimentar. Estas plataformas integran cores que exhiben diferencias significativas en cuanto a la frecuencia del procesador, microarquitectura y tamaños de cache. Sin embargo, el escaso número de cores que poseen estas plataformas reales representa una limitación a la hora de evaluar cargas de trabajo que incluyen aplicaciones paralelas con gran número de hilos. Para disponer de un mejor escenario para evaluar este tipo de cargas de trabajo, continuamos emulando la asimetría mediante la reducción de frecuencia en multicores simétricos que poseen mayor cantidad de cores.

La plataforma Intel-8 está compuesta por dos procesadores Intel quad-core (es decir, posee 8 cores en total). La emulación de cores lentos se llevó a cabo mediante una reducción conjunta de la frecuencia y el *duty cycle* del core. Específicamente, los cores lentos emulados operan en el estado DVFS de menor frecuencia en esta plataforma (2,0 GHz) y al 75 % del máximo *duty cycle*, mientras que los cores rápidos operan a la frecuencia máxima soportada (3,0 GHz) y al 100 % del *duty cycle*. La reducción conjunta de frecuencia y de *duty cycle* nos permitió obtener mayores ratios de rendimiento *fast-to-slow* (*speedup factors*) que los que pueden conseguirse reduciendo sólo la frecuencia.

La plataforma AMD-12 está compuesta por dos procesadores AMD hexa-core (es decir, posee 12 cores en total). La emulación de los cores lentos sobre esta plataforma se llevó a cabo

mediante la reducción de la frecuencia del core. Específicamente, los cores lentos operan a una frecuencia de 800 Mhz mientras que los cores rápidos operan a 2,6 Ghz.

El prototipo asimétrico *QuickIA* de Intel es un sistema UMA de doble socket que integra dos procesadores multicore: un procesador compuesto por cuatro cores de alto rendimiento (Intel Xeon E5450) y un procesador dual-core de bajo consumo (Intel Atom N330). En esta plataforma los cores de alto rendimiento (Xeon) implementan un pipeline con ejecución fuera de orden, y los cores de bajo consumo (Atom) poseen ejecución en orden. Estas diferencias a nivel microarquitectónico dan lugar a un mayor espectro de ratios de rendimiento *fast-to-slow* entre aplicaciones que el que puede conseguirse mediante asimetría en frecuencias.

La placa de desarrollo ARM Juno integra un procesador asimétrico de la familia big.LITTLE[5] de ARM, compuesto por dos grupos (clusters) de cores: uno de los clusters integra dos cores rápidos ARM Cortex A57, y el otro integra cuatro cores lentos ARM Cortex A53. Cabe destacar que a diferencia del resto de plataformas consideradas, esta placa permite obtener medidas de consumo energético desde el sistema operativo. Una FPGA (llamada IOFPGA) integrada en la placa expone al software de sistema un conjunto de registros para obtener la intensidad de corriente y el consumo de potencia instantáneos, así como el consumo energético acumulado para ambos clusters de cores Cortex A57 y Cortex A53 desde el arranque del sistema. Por otra parte, existen registros adicionales que proporcionan mediciones similares para la GPU integrada en la placa y para el resto del *system-on-chip* (SoC), que incluye el controlador de DRAM entre otros componentes.

2.2. Framework de planificación para multicores asimétricos

Por la complejidad que representa realizar modificaciones en el planificador de un SO, la mayoría de los investigadores han evaluado estrategias de planificación para AMPs utilizando simuladores [30, 8, 11, 65] o prototipos de planificación en espacio de usuario [45, 46]. Por el contrario, en esta tesis modificamos los planificadores de los sistemas operativos OpenSolaris y Linux para incluir implementaciones de varios algoritmos de planificación asimétrica.

Para lograr nuestro objetivo, diseñamos un framework de planificación cuya estructura se ilustra en la figura 2.1. El componente central del framework es el planificador de procesos del sistema operativo. A su vez, el planificador interactúa con varios componentes que ayudan a la toma de decisiones de los algoritmos de planificación. La interacción entre el planificador y los distintos componentes se describen a lo largo de la tesis.

La razón por la que implementamos este framework en dos sistemas operativos diferentes fue la evolución de OpenSolaris durante el desarrollo de esta tesis. El framework original basado en OpenSolaris y del cual partimos fue desarrollado en [51].

OpenSolaris fue creado por Sun Microsystems en 2007 a partir de una combinación de software de código abierto. En 2010, Oracle adquirió Sun Microsystems y tiempo después

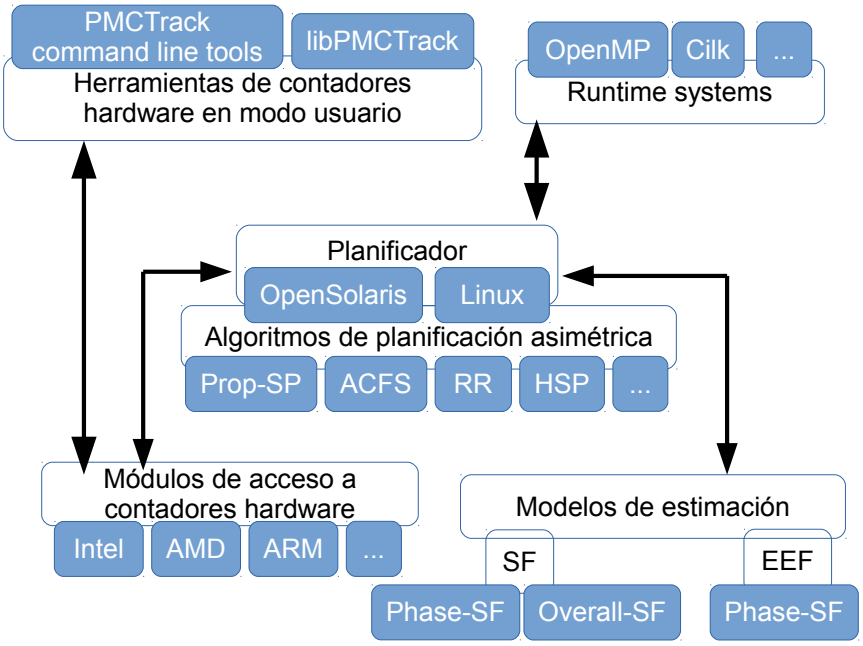


Figura 2.1: Framework asimétrico.

decidió detener el proyecto OpenSolaris y suspender tanto la distribución del sistema operativo como el modelo de desarrollo. Esto supuso un problema para nuestro framework que estaba implementado sobre éste sistema operativo, que ya no iba a dar más soporte para hardware nuevo. Asimismo, la aparición de plataformas asimétricas reales, como por ejemplo *big.LITTLE* de ARM, para la cual OpenSolaris no está disponible, supuso una importante motivación para construir un nuevo framework de planificación sobre otro sistema operativo.

Por lo anterior, el framework sobre OpenSolaris fue migrado completamente al kernel Linux con todo lo que esto significa. En general, realizar modificaciones profundas en un componente central del kernel de un sistema operativo, como el planificador, resulta más complejo que el desarrollo tradicional de aplicaciones en espacio de usuario. En particular, la migración del framework superó ampliamente las 15000 líneas de código, su depuración fue de gran complejidad, y se debió prestar especial atención a una amplia y variada cantidad de aspectos relacionados a la concurrencia. Por todo ello, la migración del framework de planificación al kernel Linux demandó una gran cantidad de tiempo durante el transcurso de esta tesis doctoral.

En el resto de esta sección comenzamos describiendo las características de los planificadores de OpenSolaris y Linux, y las modificaciones realizadas para dar soporte para AMPs en estos sistemas. En el caso particular de OpenSolaris, sólo nos enfocamos en algunos aspectos críticos de la estructura del planificador, necesarios para entender en qué consistió el proceso de migración del código de OpenSolaris a Linux. Sin embargo, en esta descripción omitimos algunos aspectos de bajo nivel relacionados con el *CPU accounting*, el balance de carga, la

gestión de *timeslices* y la representación interna de los procesos. Los detalles de la implementación del framework sobre OpenSolaris, la estructura de este sistema operativo y otros aspectos de bajo nivel pueden encontrarse en [51]. Asimismo, en [67] se presentan las principales diferencias entre los planificadores de OpenSolaris y Linux. En cuanto a Linux, plataforma que utilizamos más intensivamente, describimos en detalle la estructura del planificador y las modificaciones realizadas para dar soporte a AMP. Para finalizar la sección, realizamos la descripción de los componentes restantes del framework de planificación.

2.2.1. El planificador de OpenSolaris

El kernel de OpenSolaris está compuesto por varios subsistemas. El subsistema encargado de la planificación de procesos, conocido como subsistema central o *core*, consta de dos componentes: el *dispatcher* y los *módulos de planificación*. El *dispatcher* es el encargado de encolar los procesos ejecutables en las colas de planificación o *run queues* (existe una *run queue* por CPU), seleccionar el siguiente proceso a ejecutar en un procesador y gestionar los cambios de contexto entre procesos. Los *módulos de planificación* en OpenSolaris permiten agregar nuevos algoritmos de planificación al sistema operativo. Mientras que el dispatcher se encarga de tareas clave como el balance de carga y la gestión global de las *run queues*, los *módulos de planificación* manejan el conteo de ciclos de CPU (*CPU accounting*), la expiración del *timeslice* y las prioridades.

Una estructura importante en OpenSolaris son las clases de planificación que se implementan como módulos de planificación independientes y se pueden cargar dinámicamente en el kernel. Cada proceso en el sistema pertenece a una de las múltiples clases de planificación disponibles. La clase determina el algoritmo con el cual se planificará al proceso. La clase de planificación de un proceso se hereda del proceso padre, pero puede cambiarse posteriormente mediante la llamada al sistema `priocntl()`.

La versión actual del kernel de OpenSolaris soporta cinco clases de planificación disponibles para las aplicaciones de usuario:

- Tiempo compartido (TS) - clase predeterminada -
- Interactive (IA)
- Fair Share (FSS)
- Prioridad fija (FX)
- Tiempo real (RT).

Existe además la clase SYS, la cual está reservada por el kernel para la ejecución de procesos del sistema operativo.

2.2.1.1. OpenSolaris y soporte AMP

A partir de los *módulos de planificación* es posible agregar nuevas clases que implementen algoritmos de planificación para dar soporte a AMP. Un algoritmo de planificación asimétrica debe ser capaz de realizar acciones clave como forzar que un proceso se ejecute en un tipo de core en particular (rápido o lento), o permitir que los cores rápidos reciban mayor carga que los cores lentos. Sin embargo, esto no puede realizarse a partir del código de un *módulo de planificación* convencional, ya que la interfaz entre el *dispatcher* y las clases de planificación limita las acciones que puede realizar el módulo. Por ejemplo, ciertas operaciones como seleccionar una CPU específica para un proceso o controlar la carga de un conjunto de CPUs están restringidas al código del *dispatcher*, que es el único que tiene acceso a las *run queues* por CPU.

Para superar estas dificultades se introdujeron una serie de *callbacks* en el código del *dispatcher* que permiten a los *módulos de planificación* asimétrica trabajar cooperativamente con el *dispatcher*. Específicamente, los *módulos de planificación* asimétrica son notificados cada vez que el *dispatcher* asigna una CPU a un proceso y también cuando un proceso debe migrar de un core a otro para lograr el balance de carga adecuado. Ante una notificación, el *módulo de planificación* puede seleccionar un procesador de destino diferente al elegido por el *dispatcher*, evitando así asignaciones "no deseadas" de procesos a cores, como por ejemplo: asignar un proceso a un core rápido cuando debería asignarse a un core lento.

Para diferenciar en el sistema operativo los distintos tipos de core se ideó una estrategia basada en particiones. En particular, todos los cores en el sistema se organizan en particiones, es decir conjuntos disjuntos de cores formados por cores rápidos y cores lentos. El planificador asimétrico realiza asignaciones de procesos a particiones (conjuntos de cores) en lugar de hacerlo a cores individuales.

Para implementar estas particiones, utilizamos la abstracción de *processor sets* del kernel de OpenSolaris. Los *processor sets* permiten al administrador dividir los cores de un sistema multicore en distintos grupos. Todos los procesos tienen asignado un *processor set*, que se hereda del proceso padre, pero puede cambiarse posteriormente a través de herramientas administrativas.

En nuestra implementación, creamos un *processor set* por cada partición de cores: una para los cores rápidos y otra para los cores lentos. El *dispatcher* fuerza a los procesos a ejecutarse en el *processor sets* que fueron asignados y realiza el balance de carga de forma independiente en cada *processor sets*. Por lo tanto, el balance de carga está garantizado entre los cores del mismo *processor sets*, pero no entre *processor sets*. El balance de carga entre particiones depende de las políticas de los algoritmos implementados en el módulo de planificación asimétrica.

2.2.2. El planificador de Linux

El planificador de Linux tiene un diseño modular donde interactúan los siguientes componentes:

- Las clases de planificación.
- Las colas de procesos listos para ejecutar o *run queues*.
- Las estructuras de datos para representar procesos y entidades.
- El *core scheduler* conformado por dos componentes:
  - El planificador periódico.
  - El planificador principal.

A continuación describimos en detalle la estructura del planificador de Linux y cada uno de sus componentes. En particular, pondremos especial énfasis en la clase de planificación *Completely Fair Scheduler*, utilizada como base para la implementación de nuestro framework de planificación asimétrica sobre Linux.

2.2.2.1. Las clases de planificación

En el planificador de Linux existen varias clases de planificación. Una clase de planificación implementa un algoritmo de planificación particular. De esta manera, el planificador de Linux permite que coexistan distintas políticas de planificación de procesos. Cada proceso en el sistema operativo pertenece a una única clase de planificación y cada clase de planificación es responsable de gestionar sus propios procesos.

La versión actual del kernel Linux soporta cinco clases de planificación:

- **Stop (Stop class):** no está asociada a una política de planificación. Esta clase se utiliza como mecanismo para forzar la ejecución de un proceso en un procesador y así expropiarlo, o cuando se necesita desactivar un procesador.
- **Earliest deadline first algorithm (DL class):** esta clase permite a los procesos declarar una cantidad de trabajo necesario y un tiempo límite en el cual este trabajo debe ocurrir; con esto se garantiza que todos los procesos alcancen ese tiempo límite. Suele utilizarse en procesos relacionados con tiempo real.
- **Real time (RT class):** se encarga de planificar procesos de tiempo real, es decir, procesos que tienen restricciones de tiempo importante. Linux provee dos políticas de planificación de tiempo real:

    - **SCHED_FIFO:** un proceso que sigue esta política se ejecutará hasta que se bloquee o deje el procesador explícitamente.
    - **SCHED_RR:** el procesador se comparte por un período de tiempo determinado entre los procesos de la misma prioridad.

- **Completely Fair Scheduler (CFS class):** es la encargada de planificar los procesos por defecto. Si se crea un proceso y no se especifica a qué clase pertenece entonces será planificado por esta clase.
- **Idle (Idle class):** no está asociada a una política de planificación. Se ejecuta cuando en un procesador no hay ningún proceso ejecutándose.

La clase de planificación de un proceso se hereda del proceso padre, pero puede cambiarse posteriormente. Es posible asignar una clase de planificación a un proceso mediante la llamada al sistema `sched_setscheduler()`.

Todas las clases de planificación realizan acciones comunes: elegir el próximo proceso a ejecutar, seleccionar el procesador donde un proceso debe ejecutar, encolar o desencolar un proceso de una *run queue*. Sin embargo, cada clase debe implementar este comportamiento de acuerdo a su propia política.

Internamente, cada clase de planificación se representa como una estructura (*struct sched_class*). Dentro de esta estructura se encuentran referencias a funciones que implementan el comportamiento propio de la clase. A su vez, las clases están ordenadas según su prioridad. Para mantener este orden, la estructura de clase posee un enlace a la clase siguiente (de menor prioridad), formando entre las clases una lista. En Linux, la lista de clases la encabeza la clase Stop siendo la de mayor prioridad. Luego, la siguen DL, RT y CFS, en ese orden. Al final de la lista se encuentra la clase IDLE siendo esta la de menor prioridad.

2.2.2.2. Run queues

Todo proceso que está listo para ejecutar se almacena en una estructura conocida como *run queue*. Existe una *run queue* por cada procesador y cada una se divide a su vez en varias *run queues*, una por cada clase de planificación. Esta organización jerárquica (que se ilustra en la figura 2.2) permite a las clases de planificación organizar sus procesos en la estructura de datos que mejor se adapte a las políticas que cada una implemente. Como ejemplo de esto, la clase RT utiliza una lista como estructura de datos para organizar sus procesos en la *run queue*, mientras que la clase CFS utiliza una estructura conocida como *red-black tree*.

Linux implementa la *run queue* específica de un procesador en una estructura llamada *rq*. Esta estructura almacena varias características, como por ejemplo: las referencias a las *run queues* específicas de cada clase; la cantidad total de procesos encolados en ese procesador; y el proceso que actualmente se encuentra en ejecución. Internamente este proceso se conoce como *current*.

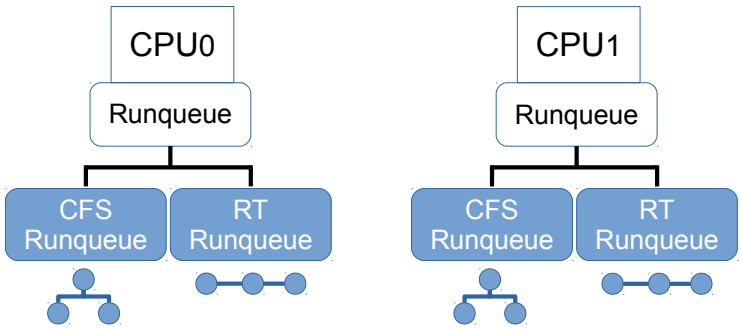


Figura 2.2: Organización de las run queues en Linux.

2.2.2.3. Representación de procesos en Linux

En Linux un proceso se representa por una estructura llamada *task_struct*. En el caso particular de las aplicaciones multi-hilo, existe un *task_struct* para representar cada hilo de la aplicación. Asimismo, otros procesos que no son procesos de usuario, como los *kernel threads*, se representan también mediante un *task_struct*.

La estructura *task_struct* almacena toda la información relacionada a un proceso o hilo, como por ejemplo: prioridades, distintos flags, el identificador de proceso o pid, la clase de planificación a la cual el proceso pertenece, el estado del proceso, y otras características. Ademas, el *task_struct* hace referencia a las entidades.

Una entidad es una estructura que almacena características de un proceso específicas de la clase de planificación a la que éste pertenece. Las clases CFS, RT y DL definen la estructura de entidad para sus procesos. Por ejemplo: la entidad de la clase CFS se representa mediante la estructura *sched_entity*; la entidad de la clase RT se representa mediante la estructura *sched_rt_entity*; y la entidad de la clase DL se representa mediante la estructura *sched_dl_entity*.

2.2.2.4. Core scheduler

El planificador de Linux puede activarse en dos situaciones:

1. Por un mecanismo periódico que se activa con cierta frecuencia verificando si es necesario realizar algún cambio de contexto.

2. Cuando un proceso se bloquea, termina su ejecución o libera el procesador por cualquier otra razón.

La funcionalidad asociada a las activaciones del planificador se implementa en el *Core Scheduler* de Linux, que consta a su vez de dos componentes: el *planificador periódico* y el *planificador principal*. A continuación describimos las principales características de ambos.

El planificador periódico

El planificador periódico está implementado en la función `scheduler_tick()`. Esta función se invoca con cierta frecuencia por cada core mientras el sistema se encuentra en actividad. Por ejemplo, en Linux y para arquitecturas x86 se establece por defecto un intervalo de 4 milisegundos entre dos invocaciones consecutivas de la función en una misma CPU. Sin embargo, cuando el sistema se encuentra ocioso, la función se invoca con menor frecuencia o puede incluso desactivarse totalmente con el fin de ahorrar energía. La función `scheduler_tick()` lleva a cabo las siguientes tareas:

- Gestiona estadísticas específicas del planificador relacionadas a procesos/hilos individuales y a todo el sistema en su conjunto.
- Activa el planificador periódico de la clase de planificación correspondiente al proceso *current*.
- En arquitecturas multicore, activa el balance de carga si es necesario.

En cuanto a la gestión estadística, la función `scheduler_tick()` modifica contadores que permiten luego tomar decisiones de planificación a nivel global o a nivel de una clase específica.

La activación del planificador periódico, específico de la clase de planificación del proceso *current*, se realiza invocando a la función `task_tick()`. Cada clase implementa esta función de acuerdo a su propia política. Al invocarse realizará las acciones que considere necesarias para sus procesos. En particular, la clase de planificación puede determinar que el proceso *current* debe abandonar el procesador, en ese caso se activará el planificador principal que es el encargado de realizar el cambio de contexto.

En arquitecturas multicore es necesario mantener la carga entre los cores lo más balanceada posible. Para este fin, la función `scheduler_tick()` determina si es necesario activar el balance de carga entre los cores, en ese caso se realizarán las acciones que correspondan. El balance de carga puede ser global o específico de una clase. Nos referimos al balance de carga global cuando involucra a todos los procesos del sistema independientemente de la clase de planificación a la cual pertenecen. En cambio, el balance de carga específico de una clase sólo involucra procesos de esta clase. En este último caso, el balance de carga es responsabilidad de cada clase de planificación. Más adelante describimos cómo se realiza el balance de carga en el caso particular de la clase CFS.

El planificador principal

El planificador principal se implementa en la función `schedule()`. Esta función se invoca en distintos puntos del código del kernel Linux y es la encargada de realizar los cambios de contexto entre los procesos. Al retornar de una excepción o de una interrupción, se comprueba si el proceso *current* tiene activado el *flag TIF_NEED_RESCHED*. Si esto es así, se invoca la función `schedule()`. Los procesos que tienen activado este *flag* son aquellos a los cuales eventualmente se debe expropiar el procesador. Este *flag* no se activa directamente, su modificación se realiza a través de una función que indica al procesador que debe replanificar (*resched*) el proceso *current*. Los desarrolladores del kernel Linux han modificado el nombre de esta función a lo largo de las distintas versiones. Según la versión su nombre puede ser `resched_task()`, `resched_cpu()` o `resched_curr()`, pero básicamente el objetivo es el mismo.

La función `schedule()` realiza los siguientes pasos:

1. *Si es necesario desactiva el proceso current (conocido en la función `schedule()` como prev).*
2. *Elige el próximo proceso a ejecutar (conocido en la función `schedule()` como next).*
3. *Realiza el cambio de contexto (cambia prev por next).*

Para desactivar un proceso en Linux se utiliza la función `deactivate_task()`. Esta función se invoca desde varios puntos del código del kernel Linux, uno de ellos es la función `schedule()`, y se comporta de dos maneras diferentes dependiendo de si el proceso es *current* o no. Si `deactivate_task()` se ejecuta sobre el proceso *current* entonces este proceso ya no podrá seguir su ejecución. En cambio, si se ejecuta sobre un proceso que no es *current*, lo habitual es que ese proceso se encuentre encolado en la *run queue*, por lo tanto el comportamiento de `deactivate_task()` es desencolarlo. En el contexto de la función `schedule()`, desactivar *prev* es invocar la función `deactivate_task()` sobre el proceso *current*. Desactivar *prev* no siempre es un paso necesario, sólo se lleva a cabo si *prev* debe dejar el procesador por alguna razón. Independientemente del proceso sobre el cual se invoque (ya sea *current o no*), la función `deactivate_task()` actualiza algunos parámetros estadísticos. Luego, delega la responsabilidad de desactivación a la clase de planificación a la cual el proceso pertenece. Para esto invoca a la función `dequeue_task()` específica de la clase del proceso, que realizará las acciones correspondientes según la política de planificación de la clase.

La elección del próximo proceso a ejecutar se realiza invocando a la función `pick_next_task()`. Esta función recorre la lista de clases de planificación, comenzando

desde la clase con mayor prioridad, hasta encontrar una clase que le retorne un proceso. Específicamente, cuando la función solicita un proceso a una clase, ésta puede retornar un proceso o bien no tener ningún proceso listo para ejecutar, en este caso se avanzará a la clase siguiente. Si ninguna clase de planificación puede proporcionar un proceso, la última clase de planificación de la lista es la clase *Idle*(la de menor prioridad), que siempre retorna un proceso conocido como *idle task*.

Si *prev* no se desactiva, la función `pick_next_task()` podría haber elegido nuevamente al proceso *prev* ($next = prev$) por lo tanto no será necesario ningún cambio de contexto. Si por el contrario, el resultado de la función`pick_next_task()` es un nuevo proceso ($prev \neq next$), entonces se realizará un cambio de contexto a nivel de *hardware* que se lleva a cabo invocando a la función `context_switch()`.

2.2.2.5. La clase de planificación CFS

La clase de planificación CFS se basa en la idea de un procesador multitarea ideal. En este procesador, cada proceso activo recibe una porción de la potencia de cómputo total permitiendo que estos se ejecuten físicamente en paralelo. Si en el sistema existen dos procesos, entonces ambos correrán al mismo tiempo, cada uno utilizando el 50 % de la potencia de cómputo total. Si estos dos procesos comienzan su ejecución en el mismo instante, el tiempo de ejecución (*runtime*) de cada proceso en cualquier momento es el mismo.

En un procesador ideal ningún proceso debe esperar. Asumiendo que los dos procesos anteriores requieren 10 minutos cada uno para completar su trabajo, la ejecución de ambos requerirá de 20 minutos, en vez de 10 minutos. Sin embargo, los dos procesos finalizarán su tarea exactamente después que este tiempo se alcance, y en ningún momento habrán estado inactivos.

Sin embargo, lo anterior no es posible en un *hardware* real con un sólo procesador donde a lo sumo puede ejecutarse un proceso por vez. Esto significa que si existen dos procesos activos, sólo uno de ellos podrá ejecutar y por lo tanto recibirá el 100 % de la potencia de cómputo total, mientras que el otro proceso deberá esperar recibiendo 0 % de la potencia de cómputo.

Para lograr una ejecución multitarea, los sistemas operativos permiten que los procesos se ejecuten por un período corto de tiempo realizando cambios de contexto una y otra vez de manera frecuente. Para los usuarios se crea la ilusión de ejecución paralela cuando en realidad no lo es. Mientras más procesadores haya en el sistema habrá una ejecución paralela perfecta, pero siempre existe el caso en que haya más procesos que procesadores y el problema se vuelve a manifestar. Alcanzar un comportamiento similar al de un procesador ideal requiere que los procesos ejecuten por un período muy corto de tiempo, pero esto es físicamente imposible de manejar debido a los costos de los cambios de contexto que lo hacen altamente

ineficiente.

CFS intenta aproximar este comportamiento ideal. Para esto hace un seguimiento de la porción de potencia de cómputo que debería tener disponible cada proceso en el sistema. Una de las características más importantes de CFS es que no requiere del concepto de *timeslice* (en el sentido tradicional). Los planificadores clásicos calculan los *timeslices* para cada proceso en el sistema y le permiten correr hasta que sus *timeslices* expiren; cuando todos los *timeslices* expiraron entonces el sistema deberá recalcularlos. En vez de esto, CFS introduce para cada proceso en el sistema el concepto de *virtual runtime*.

El *virtual runtime* se utiliza para aproximar el comportamiento del "procesador multitarea ideal" que CFS modela. Para aproximarse a la ejecución en un procesador ideal, el *virtual runtime* de todos los procesos debería ser el mismo. CFS intenta mantener el balance total entre todos los procesos activos todo el tiempo.

Cada proceso perteneciente a la clase CFS tiene asociada una variable *vruntime* que representa su *virtual runtime*. Esta variable lleva la cuenta del tiempo de ejecución actual (*actual runtime*) del proceso ponderado por el número de procesos en ejecución. A diferencia de la gestión tradicional de timeslices, el *virtual runtime* se actualiza por cada *tick* pero solamente en el proceso *current*. Esto significa que después de algún tiempo el *virtual runtime* de *current* se habrá incrementado lo suficiente, y eventualmente habrá otro proceso con un *virtual runtime* menor esperando para utilizar el procesador. En este caso, será necesario un cambio de contexto.

CFS *run queues*

En el caso particular de CFS, la clase implementa la *run queue* específica utilizando una estructura de datos organizada en forma de árbol conocida como *red-black tree*. Los procesos en la *run queue* están ordenados por su *virtual runtime*. En la entrada más izquierda del árbol se encuentra el proceso con el menor valor de *virtual runtime*, es decir, el proceso que menos tiempo ejecutó y por lo tanto el que lleva mayor tiempo esperando. Eventualmente este proceso será el elegido para ejecutar.

En particular, la estructura *red-black tree* es un árbol de entidades, es decir, no almacena la estructura que representa al proceso (*task_struct*) sino la estructura de entidad del proceso. En este caso será la estructura de entidad correspondiente a la clase CFS, nos vamos a referir a esta entidad como entidad CFS.

Entidades CFS

Una entidad en la clase de planificación CFS se representa con una estructura *sched_entity*.

Esta estructura está compuesta por campos que representan características del proceso propias de la clase, como por ejemplo: el *virtual runtime*, distintos parámetros estadísticos correspondientes a la clase y distintas referencias asociadas a la estructura *red-black tree*, entre otras.

CFS y el planificador periódico

Por cada *tick* en el sistema se invoca la función `scheduler_tick()`, la cual activa el planificador periódico de la clase de planificación del proceso *current*. Esta activación se lleva a cabo invocando a la función `task_tick()` de la clase del proceso *current*. En el caso que *current* pertenezca a la clase CFS, se delega la ejecución a la función *task_tick_fair()* de esa clase. En esta función se incrementa el *virtual runtime* del proceso. Esto significa que después de algún tiempo habrá otro proceso con un *virtual runtime* menor almacenado en la *run queue*, y será necesario expropiar el procesador a *current* en favor del proceso con menor *virtual runtime* de la *run queue*.

Si el proceso *current* ha ejecutado el tiempo que merece, entonces se invoca una función de replanificación (*resched*) que activa el *flag TIF_NEED_RESCHED* en este proceso. Dado que marcar este *flag* implica que se active al planificador principal, una vez terminada la llamada a `scheduler_tick()` y en el momento que se considere adecuado, se invocará a la función `schedule()` que se encargará de expropiar el proceso.

CFS y el balance de carga

En arquitecturas multicore, el algoritmo de planificación CFS intenta distribuir los procesos que pertenecen a su clase de forma balanceada entre los procesadores. Para lograr este objetivo aprovecha el scheduler periódico. Por cada llamado a la función `scheduler_tick()`, independientemente de la clase del proceso *current*, al finalizar se invoca a la función `trigger_load_balance()`.

Es importante mencionar, que si bien la función `trigger_load_balance()` se invoca por cada *tick*, en realidad no se realiza un balance de carga con esta frecuencia, dado que esto podría afectar significativamente el rendimiento del sistema. El período en el cual el próximo balance de carga debe realizarse se calcula cada vez que termina el balance de carga actual en función del estado del sistema. Por lo tanto, la función `trigger_load_balance()` comprueba si transcurrió ese período de tiempo, y si es así, activa una interrupción por software (SOFTIRQ) diferida que será la encargada de invocar las funciones que realizarán el balance de carga.

La función principal que realiza el balance de carga es `load_balance()`. Esta función recorre las *run queues* de la clase CFS y determina cuál es la *run queue* con mayor carga (*busiest*).

Luego, intenta migrar procesos desde esta *run queue* hacia la *run queue* del procesador donde la función `load_balance()` fue invocada.

Es importante mencionar que el balance de carga activado en un procesador no realiza movimientos de procesos entre otros procesadores, sino que los movimientos siempre se dan desde el procesador con mayor carga hacia el procesador donde el balance se activó, siguiendo una estrategia "*work stealing*". Siguiendo esta estrategia se tiene una implementación eficiente del balance de carga. Dado que la SOFTIRQ puede activarse en varios procesadores al mismo tiempo, es posible balancear la carga en más de un procesador a la vez.

Cabe destacar, que el período de tiempo en el cual se debe llevar a cabo el balance de carga no es único, debe gestionarse un período por cada procesador y cada período es totalmente independiente del otro.

CFS y procesadores ociosos

Una característica particular de CFS es que toma ciertas decisiones de planificación cuando en el sistema queda un procesador ocioso. Al activarse el planificador principal sobre un procesador (recordemos que esto desencadena una llamada a la función `schedule()`), existe la posibilidad de que el proceso *current* libere el procesador y la *run queue* de ese procesador se encuentre vacía, es decir, no existen otros procesos para ser ejecutados. Ante esta situación, y antes de ser invocada la función `pick_next_task()`, CFS intenta traer procesos de algún procesador con mayor carga, siguiendo la misma estrategia "*work stealing*" del balance de carga. Esto se conoce en Linux como *idle balance* y su comportamiento está implementado en la función `idle_balance()`. Si tiene éxito trayendo algún proceso desde otro procesador, entonces la *run queue* ya no estará vacía, y por lo tanto, la función `pick_next_task()` podrá elegir a este proceso para que sea el próximo en ejecutar.

2.2.2.6. Linux y soporte AMP

Si bien Linux y OpenSolaris tienen cierta similitud, existen diferencias importantes que hicieron que portar el framework de planificación asimétrica implementado en OpenSolaris hacia Linux no fuera directo.

A diferencia de Solaris, las clases de planificación en Linux no se implementan como módulos de planificación que pueden cargarse dinámicamente en el kernel. Por el contrario, las clases se implementan de forma estática en el código del kernel. Por esta razón, agregar una nueva clase de planificación o realizar modificaciones en alguna clase existente, implica recompilar el kernel Linux, lo cual representa una importante cantidad de tiempo.

Otra de las diferencias, es que en el kernel Linux no existe la abstracción de *processor sets* del kernel de OpenSolaris. Por lo tanto, se debió emplear otro mecanismo para crear las

particiones de cores. En nuestra implementación utilizamos máscaras de afinidad y se creó una máscara por cada partición de cores: una para los cores rápidos y otra para los cores lentos. Una máscara de afinidad es un conjunto de bits donde cada bit está asociado a un core. Un bit con valor 1 en la máscara de afinidad de una partición, significa que el core asociado a ese bit pertenece a esa partición. Por el contrario, si el bit tiene valor 0 entonces el core asociado no pertenece a esa partición. Existe una restricción y es que un bit no puede tener valor 1 en ambas particiones. Sin embargo, es posible que un bit tenga valor 0 en ambas particiones, esto significa que el core asociado está deshabilitado. Una ventaja de usar máscaras de afinidad es que éstas pueden modificarse sin necesidad de reiniciar el sistema.

Aunque las máscaras de afinidad nos permitieron crear las particiones, no poseen las ventajas que tienen los *processor sets* de OpenSolaris. En particular, la ausencia de *processor sets* en el kernel Linux afecta al balance de carga. En OpenSolaris el *dispatcher* garantiza el balance de carga entre los cores de la misma partición (mismo *processor sets*) y los algoritmos implementados en el módulo de planificación asimétrica garantizan el balance de carga entre particiones (entre *processor sets*). En Linux el balance de carga entre particiones también está garantizado por los algoritmos de planificación asimétrica. Sin embargo, Linux no puede garantizar el balance de carga entre cores de una misma partición creada con máscaras de afinidad porque no está implementado. Por esta razón, fue necesario añadir código específico en nuestro *framework* de planificación para dar soporte al balance de carga dentro de una misma partición.

Cuando se quiere ejecutar una aplicación bajo una política de planificación asimétrica es necesario enviar información al sistema operativo. Para esto creamos un *lanzador*. Un *lanzador* es un *script* que nos permite ejecutar aplicaciones de usuario bajo una política de planificación para AMP. Este *script* entre otras cosas selecciona la clase de planificación de la aplicación. La llamada al sistema `priocntl()` de OpenSolaris, que permite cambiar de clase de planificación a un proceso, no está disponible en Linux. Sin embargo, existe un mecanismo similar para asignar una clase de planificación a un proceso mediante la llamada al sistema `sched_setscheduler()`. La existencia de una llamada al sistema diferente nos obligó a desarrollar un *lanzador* específico de cada sistema operativo para llevar a cabo nuestros experimentos.

Existen otras diferencias como las relacionadas al *CPU accounting*, la gestión de *timeslices*, el balance de carga y la representación interna de los procesos. Estas diferencias pueden encontrarse en [67].

Para poder implementar sobre Linux algoritmos de planificación asimétrica agregamos una nueva clase de planificación basada en la clase de planificación CFS y que llamamos clase AMP. Implementamos la clase AMP con menor prioridad que la clase CFS, ubicándola después de ésta en la lista enlazada de clases. Fue importante aprovechar las ventajas de la clase CFS dado que ofrece un buen comportamiento en las arquitecturas simétricas actuales. Sin embargo, su implementación no está preparada para los desafíos que proponen las

arquitecturas asimétricas. Esencialmente, la clase AMP se encarga de la planificación entre las distintas particiones de cores, y delega a CFS la responsabilidad de planificar procesos dentro de la misma partición.

La clase AMP y CFS tienen algunos aspectos en común, como por ejemplo, el hecho de que ambas utilizan la estructura de datos *red-black tree* para implementar las *run queues*. Sin embargo, la clase AMP agrega nuevas características para dar soporte asimétrico:

- La estructura para representar entidades específicas de la clase AMP
- La estructura para representar aplicaciones (como conjunto de hilos)
- Soporte para múltiples algoritmos de planificación en la misma clase
- Migración de hilos entre particiones
- Interacción con el planificador principal
- Implementaciones específicas de balance de carga y de *idle balance*
- Interacción con el planificador periódico

A continuación describimos en detalle cada una de estas características y otros aspectos relacionados.

Entidad AMP

Para almacenar información específica de los procesos que pertenecen a la clase AMP fue necesario agregar una nueva estructura de entidad. La clase de planificación AMP define para sus procesos una nueva estructura de entidad llamada *sched_amp_entity*. Entre los campos específicos que almacena ésta estructura se encuentran: el tipo de core en el que se encuentra el proceso o hilo, *flags*, la aplicación a la cual el hilo pertenece, ciertos parámetros estadísticos, entre otros. Fue necesario además agregar a la estructura *task_struct* un nuevo campo que haga referencia a la nueva entidad que utilizarán los procesos que pertenecen a la clase de planificación AMP.

Representación de aplicaciones

Linux considera los procesos o hilos individualmente. Dicho de otra manera, los hilos que corresponden a una misma aplicación paralela, tales como las implementadas con *Pthreads* u *OpenMP*, son tratados en forma independiente respecto a la aplicación a la cual pertenecen. Aunque es posible determinar si un hilo pertenece a una aplicación mediante la relación

padre-hijo establecida a partir de *flags*, los hilos no están agrupados en una estructura común. Si bien existen en Linux formas de agrupamiento, la manera de hacerlo no es transparente al usuario y debe ser éste el encargado de generar tal agrupamiento.

Algunos de los algoritmos de planificación sobre multicores asimétricos evaluados en esta tesis, necesitan tomar decisiones a nivel de aplicación, es decir, considerando todos los hilos que la componen. Para esto es necesario agrupar aquellos hilos que pertenecen a la misma aplicación en alguna estructura a nivel de kernel.

Para tal fin, incorporamos a la clase AMP una estructura de aplicación que permite agrupar los hilos de una misma aplicación. En particular, un hilo pertenece únicamente a una aplicación y una aplicación puede estar formada por uno o más hilos. A su vez la clase AMP mantiene dos listas: una que almacena aplicaciones multi-hilo y otra que almacena aquellas aplicaciones secuenciales. Se considera aplicación secuencial a aquélla que posee un sólo hilo, ya sea porque es una aplicación secuencial propiamente dicha o porque es una aplicación paralela que por distintas circunstancias sólo tiene un hilo en ejecución.

Representamos una aplicación en la clase de planificación AMP con una estructura que llamamos *application*. Esta estructura está compuesta por campos que representan las características de la aplicación, como por ejemplo: la lista de hilos activos de la aplicación, un identificador de la aplicación, distintos campos estadísticos, entre otras.

Para la construcción de la estructura de aplicación, la clase AMP utiliza los *flags* de creación de procesos de Linux que dan información de la relación padre-hijo entre estos. Cada vez que se crea un proceso o hilo en Linux se invoca a la función `sched_fork()`. Esta función recibe como parámetro un conjunto de *flags*. A partir de estos *flags* es posible determinar quién es el proceso padre del hilo creado, y en consecuencia si el hilo creado es independiente o perteneciente a una misma aplicación de usuario.

En Linux existen dos tipos de llamadas al sistema que permiten generar procesos o hilos. La primera es mediante `fork()` y la segunda utilizando `clone()`. La llamada a `fork()` genera una copia idéntica e independiente del proceso padre. La llamada a `clone()` funciona de forma similar a `fork()` pero la copia no es independiente del proceso padre ya que pueden compartir recursos. Los *flags* que se reciben como parámetro en la función `sched_fork()` permiten determinar la forma de creación de los procesos.

Cuando se crea un proceso o hilo perteneciente a la clase AMP, se evalúan los *flags* y se determina el tipo de creación (*fork* o *clone*). Si la creación fue mediante `fork()` entonces se corresponde a un proceso independiente, por lo tanto debe crearse una nueva estructura de aplicación. Si en cambio, la creación fue mediante `clone()`, AMP entiende que se creó un hilo de una aplicación multi-hilo, y entonces el hilo hereda la estructura de aplicación asociada al padre.

Soporte para múltiples algoritmos de planificación

En esta tesis evaluamos distintos algoritmos de planificación asimétrica. Por la complejidad y el tiempo que demanda implementar una nueva clase de planificación descartamos la idea de crear una clase de planificación independiente para cada algoritmo. En nuestro caso, decidimos desarrollar la clase de planificación AMP de una forma genérica. Esto permite implementar en la misma clase varios algoritmos de planificación.

Desde modo usuario sólo puede estar activo un algoritmo de planificación asimétrica en un instante determinado. Sin embargo, es posible cambiar de un algoritmo a otro desde modo usuario mediante un parámetro de un módulo de kernel, sin necesidad de reiniciar el sistema. Es importante mencionar que el hecho de cambiar de algoritmo de planificación afecta sólo a las aplicaciones que pertenecen a la clase AMP.

Consideraciones en cuanto a las decisiones de planificación asimétrica

Antes de continuar con la descripción de los componentes de la clase AMP realizaremos algunas consideraciones con respecto a las decisiones sobre planificación en la nueva clase.

Uno de los desafíos en la implementación de la clase AMP es determinar en qué momento se deben tomar ciertas decisiones de planificación. Por ejemplo: ¿Cuándo se debe migrar un hilo de un tipo de core a otro?, ¿Cuándo y cómo se deben modificar ciertos parámetros que afectan a la planificación sobre AMP?.

La mayoría de las decisiones de planificación en nuestra clase AMP se toman en el momento que un hilo abandona el sistema o ingresa al mismo. Otras decisiones de planificación se llevan a cabo en el planificador principal (función `schedule()`), en particular las relacionadas a migraciones e *idle balance*, y en el planificador periódico (`scheduler_tick()`) aquellas relacionadas con *CPU accounting* y al balance de carga. Estos dos últimos puntos los describiremos en detalle más adelante. Por ahora sólo nos centraremos en las decisiones a tomar cuando los hilos abandonan o ingresan al sistema.

Cuando un hilo abandona el sistema, en este caso vamos a decir que el hilo se desactiva, se libera el procesador y éste podrá ser utilizado por cualquier otro hilo. Las razones por las cuales un hilo puede desactivarse son: terminó su ejecución, necesita realizar una operación de entrada-salida, o deja de ejecutar por cuestiones relacionadas con sincronización.

Por otro lado, un hilo ingresa al sistema, en este caso decimos que el hilo se activa, en la siguientes situaciones: el hilo fue recién creado, retorna luego de hacer una operación de entrada salida, o ”estaba dormido” por algún mecanismo de sincronización y fue ”despertado”. Se debe tener en cuenta, que un hilo que se activa no necesariamente se ejecutará de forma inmediata, simplemente se encolará en la *run queue* y esperará por su turno para usar el

procesador.

El único caso en que no se toman decisiones críticas de planificación es cuando los hilos se activan o desactivan por una migración.

Las funciones para activar o desactivar hilos son propias de Linux y se invocan por distintas razones en varios puntos del kernel. En Linux, un hilo se activa mediante la función `activate_task()`. Por el contrario, un hilo se desactiva con la función `deactivate_task()`. Si bien estas funciones se llaman cuando un hilo ingresa o abandona el sistema también se invocan en otros contextos y con distintos efectos. Anteriormente, mencionamos como la función `deactivate_task()` se comporta de forma diferente dependiendo del hilo sobre el cual se invoca. Recordemos, que si el hilo es *current* entonces detiene su ejecución, pero si el hilo no es *current* se lo desencola de la *run queue* donde se encuentra encolado. Algo similar ocurre con `activate_task()`, que se utiliza durante la migración de hilos para encolar en las *run queues* hilos que han sido migrados desde otro procesador.

Las funciones `activate_task()` y `deactivate_task()` reciben como parámetros *flags* que indican las razones por las cuales se invocaron. Modificamos estas funciones para hacer uso de los *flags* y en función de ellos determinar si es necesario tomar alguna decisión de planificación para la clase AMP. Específicamente, cuando se invoca `activate_task()` o `deactivate_task()` para un hilo de la clase AMP, se invoca a las funciones `amp_active()` y `amp_inactive()`, respectivamente. Estas funciones realizan distintas acciones. Dado que la clase AMP implementa varios algoritmos de planificación, toda decisión tomada dentro de las funciones `amp_active()` o `amp_inactive()` depende del algoritmo de planificación actual.

Para ejemplificar, supongamos que un hilo de la clase AMP que se está ejecutando en un core rápido termina su ejecución, y que los hilos restantes están ejecutándose sobre cores lentos. El evento de terminación desencadena una invocación a la función `deactivate_task()`. Nuestro código dentro de la función lee los *flags*, y como el valor de estos indican que debe ser tratado por la clase AMP se invoca a la función `amp_inactive()`. La mayoría de los algoritmos de planificación para AMP intentan optimizar el uso de los cores rápidos, por lo tanto cuando `amp_inactive()` detecta que el hilo terminó su ejecución y que el core rápido queda libre, intenta traer un hilo que se encuentre ejecutando en un core lento. Sin embargo, sólo se indica que ese hilo debe migrarse de un core lento a un core rápido; la migración efectiva no puede realizarse en este punto del código.

En algunas ocasiones, cuando se toman ciertas decisiones en un lugar determinado del código, las acciones asociadas no pueden llevarse a cabo inmediatamente sino que deben postergarse hasta un momento más adecuado o más seguro. Esto se debe a distintas razones relacionadas al rendimiento, a la concurrencia (sincronización) o al contexto en el cual se está ejecutando el hilo (kernel o interrupción). En nuestra implementación el caso más relevante donde se da esta situación es en las migraciones de hilos, que explicamos a continuación.

Migración de hilos entre particiones

Para lograr una distribución eficiente de los ciclos de cores rápidos, la mayoría de los algoritmos de planificación asimétrica realizan migraciones de hilos entre los distintos tipos de core. Cuando se decide migrar un hilo de un tipo de core a otro, las acciones asociadas a esta migración no pueden ejecutarse en ese momento sino que deben demorarse hasta un punto más adecuado o seguro del código. Entonces, nos planteamos dos interrogantes:

1. ¿En qué momento la clase AMP decide si se requiere una migración?
2. ¿Cuándo puede hacerse efectiva esa migración?.

Para responder al primer interrogante, la clase AMP decide si se requiere migrar uno o varios hilos entre los distintos tipos de cores cuando se activa o desactiva un hilo en el sistema. Éste no es el único caso, eventualmente pueden tomarse decisiones de migración durante la ejecución del planificador periódico (`scheduler_tick()`).

Cuando la clase AMP determina que un hilo debe migrarse, ”marca” ese hilo y lo agrega en una lista de migración. Por cuestiones de diseño, implementamos en la clase AMP una lista de migración para cada tipo de core. Siguiendo con el ejemplo anterior, cuando un hilo está ejecutándose en un core lento y el algoritmo de planificación determina que debería correr en un core rápido, el hilo se ”marca” para migrar a un core rápido y se agrega en una lista de migración de core rápido.

Para ”marcar” un hilo que debe ser migrado, nuestra implementación hace uso de *flags* y de las máscaras de afinidad de los hilos. En particular, la máscara de afinidad de un hilo determina en qué procesadores tiene permitido correr. Por esto, la máscara de afinidad del hilo coincide con la máscara asociada a la partición de cores donde éste debe correr. Por lo tanto, ”marcar” el hilo para migrar implica (1) escribir un *flag* del hilo que indica la necesidad de migración, (2) modificar su máscara de afinidad haciéndola corresponder con la máscara de afinidad de la partición de cores a la que debe migrar. Cabe destacar, que al determinar una migración no se indica a qué procesador o core en particular debe migrarse el hilo, sólo se especifica en qué tipo de core debe correr.

Otro aspecto a tener en cuenta, es que al cambiar la máscara de afinidad del hilo, éste puede estar encolado en una *run queue* o ejecutándose en un procesador diferente al indicado por la máscara. Lejos de ser un problema, el hecho de cambiar la máscara de afinidad no tiene efectos inmediatos en el sistema. Linux se dará cuenta de ello si por alguna razón tiene que mover el hilo de un procesador a otro. En ese caso verifica si ese movimiento puede llevarse a cabo en función de la máscara de afinidad del hilo, y en última instancia asignará el procesador de acuerdo a su máscara.

Para responder al segundo interrogante, una migración puede realmente hacerse efectiva

en la función `schedule()` del planificador principal.

Cuando se detecta que un hilo debe ser migrado a otro tipo de core, además de registrar ese evento, se invoca a la función de replanificación (*resched*) sobre el procesador donde el hilo está ejecutándose. Esto desencadena en algún momento la invocación a la función `schedule()`.

Interacción con el planificador principal

Recordemos que el planificador principal se implementa en la función `schedule()` y que esta función es la encargada de realizar los cambios de contexto entre los hilos. Esta función realiza en orden las siguientes tareas:

> *Si es necesario desactiva el hilo current (conocido en la función `schedule()` como prev).*
> *Elige el próximo hilo a ejecutar (conocido en la función `schedule()` como next).*
> *Realiza el cambio de contexto (cambia prev por next).*

La función `schedule()` permite efectuar ciertas acciones antes de elegir un nuevo hilo y realizar el cambio de contexto, esto se conoce como *pre-schedule*. De la misma forma permite efectuar acciones posteriores al cambio de contexto, esto es llamado *post-schedule*. Así, las tareas realizadas por la función `schedule()` son:

> *Si es necesario desactiva el hilo current (conocido en la función `schedule()` como prev).*
> ***Pre-Schedule.***
> *Elige el próximo hilo a ejecutar (conocido en la función `schedule()` como next).*
> *Realiza el cambio de contexto (cambia prev por next).*
> ***Post-Schedule.***

Tanto *pre-schedule* como *post-schedule* se implementan en las funciones `pre_schedule()` y `post_schedule()`, respectivamente. Debido al diseño modular de Linux, se delega la responsabilidad de estas funciones a las clases de planificación. Cada clase sabrá cuales son las acciones que debe llevar a cabo para sus hilos tanto en `pre_schedule()` como en `post_schedule()`. La clase AMP hace uso de estas funciones e implementa en ellas el código que hace efectivas las migraciones.

Cuando el planificador principal invoca a `pre_schedule()`, determina si el hilo *current*, conocido en la función `schedule()` como *prev*, pertenece a la clase AMP. Si esto es así, invoca a la función `pre_schedule()` específica de la clase de planificación AMP.

La función `pre_schedule()` específica de la clase AMP evalúa si *prev* fue marcado para ser migrado a otro tipo de core, lo cual puede determinarse en función de los*flags* del hilo. Si

prev debe migrarse, desactiva el hilo del procesador actual para que eventualmente no pueda volver a ser elegido por la función `pick_next_task()`. Sin embargo, todavía no es posible activarlo en el otro tipo de core. Esto se debe a que *prev* aún se necesita para el cambio de contexto que se llevará a cabo posteriormente.

Una característica importante de `pre_schedule()` en la clase AMP, es la posibilidad de hacer intercambios entre hilos. Si *prev* debe migrarse, la función intenta traer un hilo desde otro tipo de core que esté marcado para migrar al tipo de core de *prev*. Como ejemplo de esto, si *prev* se encuentra en un core rápido pero fue marcado para ser migrado a un core lento, la función `pre_schedule()` desactiva *prev* y busca en la lista de migración respectiva un hilo que esté ejecutándose en un core lento pero que fue marcado para migrar a un core rápido. Si `pre_schedule()` encuentra dicho hilo, lo desactiva del core lento y lo activa en el core rápido donde *prev* se encuentra. Esta acción hace que el nuevo hilo se encole en la *run queue* del core rápido y eventualmente será el próximo en ser elegido para ejecutar. De esta forma, es posible realizar dos migraciones en un sólo paso.

Una vez realizado el cambio de contexto, se invoca a la función `post_schedule()`. En la implementación de esta función para la clase AMP, se verifica si *prev* fue desactivado para migración en `pre_schedule()`. Si esto es así, se activa *prev* en el tipo de core al que debe migrarse.

Siguiendo con el ejemplo anterior, *prev* se activaría en un core lento. Cabe destacar que si `pre_schedule()` tuvo éxito al realizar el intercambio, entonces *prev* se activará en el core lento sobre el cual se estaba ejecutando el otro hilo involucrado en el intercambio. Si por el contrario el intercambio no pudo realizarse, *prev* se activará en el core lento con menor carga.

Balance de carga - idle balance

En la clase de planificación AMP, tanto el balance de carga como *idle balance*, siguen la misma estrategia "*work stealing*" que la clase CFS.

En cuanto al balance de carga, la clase AMP actúa de forma similar a la clase CFS, aprovechando las llamadas a la función `scheduler_tick()` del planificador periódico. Por cada invocación al planificador periódico, la clase AMP evalúa si considera necesario realizar el balance de carga. Al igual que la clase CFS, el próximo balance de carga se debe realizar si transcurrió cierto período calculado en el balance de carga anterior. Dentro del planificador periódico (función `scheduler_tick()`), la evaluación se realiza en una función adaptada para la clase AMP llamada `trigger_load_balance_amp()`. Si transcurrió un período de tiempo suficiente se desencadena la activación de una SOFTIRQ (AMP_SOFTIRQ) diferida que será la encargada de invocar la función que realizará las acciones para equilibrar la carga.

La función principal de balance de carga en la clase AMP es `load_balance_amp()`. La

estrategia que utiliza es similar a la empleada por la función `load_balance()` de la clase CFS, con la diferencia que `load_balance_amp()` sólo equilibra la carga entre procesadores de la misma partición. Es decir, no realiza migraciones entre particiones. Por lo tanto no podrá mover hilos de un core rápido a un core lento y viceversa.

En cuanto a *idle balance* incorporamos una nueva función `idle_balance_amp()` que se invoca desde el planificador principal. El código de la función `schedule()` queda entonces como sigue:

> *Si es necesario desactiva el hilo current (conocido en la función `schedule()` como prev).*
> *Pre-Schedule.*
> ***idle_balance_amp.***
> *Elige el próximo hilo a ejecutar (conocido en la función `schedule()` como next).*
> *Realiza el cambio de contexto (cambia prev por next).*
> *Post-Schedule.*

De forma similar a la función `idle_balance()` de la clase CFS, esta nueva función determina si la *run queue* de la clase AMP está vacía. Si efectivamente no existe ningun proceso en la *run queue*, entonces se sigue la estrategia "*work stealing*" y la función intentará traer hilos del procesador con mayor carga. No obstante, al igual que `load_balance_amp()`, sólo traerá hilos de procesadores de la misma partición. Como se puede observar, la función para *idle balance* debe ser invocada luego del *pre-schedule* y antes de elegir el próximo hilo. Recordar que la función `pre_schedule()` es la encargada de hacer las migraciones entre particiones. La función `idle_balance_amp()` eventualmente tendrá efecto si el procesador queda ocioso y `pre_schedule()` no pudo migrar ningún hilo. Si la función `idle_balance_amp()` tiene éxito, la función que elige el próximo hilo (`pick_next_task()`) tendrá al menos un hilo para seleccionar de la *run queue*.

Interacción con el planificador periódico

Anteriormente mencionamos cómo la clase AMP interactúa con el planificador periódico en relación al balance de carga. Sin embargo, existen otros motivos que desencadenan interacciones entre la clase AMP y el planificador periódico.

Dentro de la clase de planificación AMP pueden coexistir distintos algoritmos. Cada uno de ellos puede interactuar con el planificador periódico de acuerdo a los requerimientos de su política. La mayoría de los algoritmos utilizan contadores que pueden ser incrementados o decrementados en cada llamado a `scheduler_tick()`. La modificación de estos contadores puede desencadenar distintas acciones sobre los hilos, como por ejemplo, realizar migraciones de hilos entre particiones. Al igual que ocurre con la activación y desactivación de hilos, el planificador periódico no es un lugar seguro para hacer efectivas las migraciones. Por lo

tanto, si se detecta que es necesaria una migración, se registra este evento y desencadena una replanificación para que el planificador principal haga efectiva la migración en el momento adecuado.

Si las decisiones de los algoritmos de planificación se apoyan en contadores hardware, es posible aprovechar el planificador periódico para leer los registros y en función de los valores de estos tomar decisiones de planificación, como por ejemplo: obtener una estimación del *speedup factor* de un hilo en tiempo de ejecución. Sobre esto último hablaremos en detalle en 3.2.1.1.

Clase parametrizada

La clase de planificación AMP está provista de distintos parámetros de configuración que permiten su funcionamiento. A través de estos parámetros es posible, entre otras cosas, elegir el algoritmo de planificación y modificar parámetros de estos algoritmos, configurar las particiones de cores, elegir la configuración asimétrica y habilitar o deshabilitar cores, etc. Adicionalmente, existen parámetros útiles para la depuración que permiten por ejemplo: obtener la carga de trabajo en cada partición, el número de aplicaciones en el sistema y distintos parámetros estadísticos.

Todo parámetro de configuración se encuentra en estructuras internas a la cuales es posible acceder en modo usuario a través de un conjunto de entradas */proc*. Esto le da a la clase AMP mayor flexibilidad.

2.2.3. Módulos cargables del kernel

Tanto OpenSolaris como Linux permiten ampliar su funcionalidad a través de los módulos cargables del kernel. Un módulo puede implementar un controlador de un dispositivo o servicio que puede cargarse o descargarse en el kernel del sistema operativo cuando el usuario o algún dispositivo lo solicita (dinámicamente). Esto permite agregar funcionalidad al kernel de forma dinámica.

Crear un nuevo módulo implica definir un conjunto de funciones, compilarlas y hacerlas accesibles cargando el módulo compilado. Generalmente, se utilizan para obtener o modificar información del kernel por parte de los procesos de usuario.

En el caso particular de Linux, una de las formas de utilizar estos módulos es a través de *procfs*, abreviatura de *process filesystem*, que se usa principalmente para consultar información en tiempo real sobre los procesos del sistema. *procfs* no es un sistema de archivos convencional, no consume ningún espacio de almacenamiento en disco, y sólo ocupa una limitada cantidad de memoria. *procfs* se encuentra en el sistema de archivos en el directorio */proc*. Cuando se crea un nuevo módulo del kernel, además de definir su funcionalidad, éste puede

crear entradas */proc* para interactuar con programas de usuario. Una vez cargado el módulo, las aplicaciones de usuario podrán intercambiar información con el kernel mediante esta entrada. En nuestra implementación sobre el kernel Linux hacemos uso de varios módulos accediendo a ellos a través de la entrada *procfs*.

Recordemos que en el kernel de OpenSolaris las clases de planificación se implementan utilizando módulos cargables del kernel. Sin embargo, en OpenSolaris el sistema de archivos *procfs* no se puede utilizar del mismo modo que en Linux. Por lo tanto, la interacción con módulos es más compleja [51].

Nuestro framework asimétrico explota las ventajas de los módulos cargables del kernel para agregar distintas funcionalidades. Los módulos permiten interactuar con los contadores hardware de cada arquitectura y los distintos modelos de estimación. Además, los módulos se utilizan para modificar la configuración general del planificador, elegir el algoritmo de planificación y configurar parámetros específicos de cada algoritmo.

2.2.4. Interacción con contadores hardware

Los contadores hardware (o contadores de rendimiento de hardware), conocidos por sus siglas en inglés como PMC (*Performance Monitoring Counters*), son un conjunto de registros de propósito especial incluidos en los procesadores actuales. El número de registros destinados a este fin depende de la arquitectura y del modelo de procesador. Estos registros permiten contar distintos tipos de eventos hardware como el número de fallos de caché, instrucciones por ciclo, fallos de predicción de saltos, etc.

Nuestro framework asimétrico permite al planificador interactuar con los contadores hardware disponibles en las distintas arquitecturas mediante módulos cargables del kernel. Debido a que las características de los contadores cambian de una arquitectura a otra, surge la necesidad de implementar un módulo del kernel para gestionar los PMC en cada plataforma. Durante el desarrollo realizado sobre OpenSolaris y en el desarrollo inicial sobre Linux, esto representaba una tarea compleja y el código que gestionaba los contadores hardware formaba parte del planificador. Con el tiempo esta funcionalidad fue desacoplada y permitió el desarrollo de la herramienta PMCTrack [55]. Esta herramienta proporciona un mecanismo sencillo independiente de la arquitectura para acceder a la información de los contadores desde el kernel, y permite al planificador del sistema operativo obtener datos de monitorización por hilo.

A pesar de ser una herramienta orientada al sistema operativo, PMCTrack también permite la monitorización y la recolección de datos desde espacio de usuario. De esta forma, los desarrolladores del kernel pueden realizar análisis *offline* y depuración durante las distintas fases del desarrollo del planificador. Además, la herramienta ofrece tanto al sistema operativo como al espacio de usuario, componentes para obtener información disponible en los procesadores modernos a la que no se puede acceder directamente a través de los PMCs, como por ejemplo,

el nivel de ocupación de caché que realiza una aplicación o el consumo de energía.

2.2.5. Interacción con módulos de estimación

La mayoría de los algoritmos para arquitecturas asimétricas evaluados en esta tesis, necesitan determinar métricas asociadas a la ejecución de los distintos hilos, y en función del valor de esas métricas toman decisiones de planificación. Una forma de obtener esas métricas es mediante modelos de estimación. El framework asimétrico interactúa con distintos módulos cargables del kernel que implementan los diferentes modelos de estimación para cada métrica.

En esta tesis doctoral utilizamos para una misma métrica más de un modelo de estimación. Esto se debe a que un modelo puede ser adecuado para una arquitectura pero no para otra. En la sección 6.3.2 proporcionamos más detalles acerca de este aspecto.

2.2.6. Interacción con runtime systems

Se conoce como *runtime system* al software diseñado para dar soporte a la ejecución de programas escritos en algún lenguaje de programación. En la actualidad los *runtime systems* dan soporte para la creación de aplicaciones paralelas, tal es el caso de *Pthreads* y *OpenMP*.

En el caso particular de OpenMP, el *runtime system* implementa distintas formas de distribución de iteraciones de bucles paralelos entre los hilos de una aplicación. Los planificadores utilizados en nuestro entorno experimental tienen la capacidad de interactuar con *runtime systems*. En particular, en el capítulo 4, proponemos una estrategia alternativa de distribución de iteraciones de bucles para aplicaciones OpenMP que corren sobre sistemas AMP, al cual llamamos AID. Básicamente, este tipo de interacción facilita el uso de OpenMP en una arquitectura asimétrica de manera que resulte transparente al usuario, y de esta forma no sea necesario realizar modificaciones en el código de la aplicación. La implementación de AID depende de información útil que el *runtime system* recibe del planificador. Como el *runtime system* se ejecuta en modo usuario y el planificador en modo kernel, hace falta una página de memoria compartida para lograr la interacción entre estos dos modos. Esto se puede lograr facilmente en OpenSolaris mediante *schedctl*. Linux, por el contrario, no posee esa característica y debió ser implementada.

En esta tesis usamos la implementación del *runtime system* de OpenMP integrada en el compilador GCC. En ambos sistemas operativos utilizamos la misma versión de este compilador modificada para dar soporte para AMP. Además de OpenMP, este mecanismo de comunicación entre el modo usuario y el modo kernel puede utilizarse en otros *runtime systems*.

2.3. Otras herramientas

El desarrollo y en particular la depuración a nivel de sistema operativo son tareas complejas en plataformas reales. Tampoco resulta sencillo construir los escenarios de prueba para los distintos experimentos. La diversidad en las arquitecturas, los distintos algoritmos de planificación y las diferentes suites de benchmarks empleadas, hacen que el número de parámetros a considerar y configurar sean difíciles de manejar. Afortunadamente, existen algunas herramientas auxiliares que facilitan este trabajo. Sin embargo, en algunos casos específicos fue necesario desarrollar nuevas herramientas. Las herramientas utilizadas las podemos clasificar como sigue:

- Herramientas de depuración.
- Herramientas para escenarios de prueba.

A continuación describimos las características generales de cada una de ellas.

2.3.1. Herramientas de depuración

La depuración del código del kernel de un sistema operativo es un trabajo muy complejo y requiere de una cantidad de tiempo considerable. Algunas de las características a analizar durante la depuración pueden ser: el valor de variables, el valor de parámetros de funciones o el valor de retorno de estas funciones. Sin embargo, uno de los aspectos más complejos de analizar durante la depuración son los relacionados con la sincronización, como por ejemplo: la gestión de *spinlocks* o, habilitar o deshabilitar interrupciones. Los errores relacionados con sentencias de sincronización pueden manifestarse de forma aleatoria, es decir en ciertos escenarios estos errores pueden no aparecer pero al cambiar cualquier aspecto del escenario estos errores aparecen, y en la mayoría de los casos, hacen que falle por completo el sistema operativo. Lamentablemente, no existen herramientas para prevenir estas situaciones. Por lo tanto es necesario depurar el código una vez que el error se produjo. En algunos casos, no es posible resolver la depuración con una única herramienta, y suele ser necesario utilizar una combinación de ellas.

En Linux existen herramientas de depuración como Kprobes, Systemtap, KGDB y GDB. Con Kprobes (Kernel Dynamic Probes) es posible capturar las llamadas a funciones del kernel, esto permite acceder por cada llamado al valor de los parámetros y al valor de retorno de una función en particular. Kprobes funciona como un módulo de kernel que se carga y descarga dinámicamente.

Systemtap funciona de manera similar a Kprobes en cuanto a que también captura las llamadas a funciones del kernel. Sin embargo, a diferencia de Kprobes, no se basa en módulos dinámicos, sino que proporciona una interfaz de línea de comandos y un lenguaje simple de scripting.

KGDB es un depurador del kernel Linux (y también para sistemas operativos BSD), está basado en el clásico depurador GDB, por lo tanto tiene las mismas funcionalidades. El principal inconveniente de KGDB es que necesita dos máquinas conectadas por un puerto serie RS-232, una de ellas con el depurador y otra con el kernel modificado a partir de aplicarle un parche.

Con el clásico GDB, también es posible realizar algunas tareas de depuración del kernel.

De manera similar, en OpenSolaris existen herramientas como Dtrace y MDB, utilizadas y descritas en [51].

2.3.2. Herramientas para escenarios de prueba

Los escenarios de pruebas utilizados en nuestros experimentos requieren configurar aspectos muy diversos:

- Las suites de benchmarks deben compilarse e instalarse en cada plataforma *hardware* y para cada sistema operativo en particular.
- Existen módulos del kernel que deben cargarse en el sistema y la mayoría de las veces son dependientes de la arquitectura. Cada uno de estos módulos además puede tener distintas configuraciones.
- Nuestros experimentos utilizan distitos algoritmos de planificación los cuales poseen distintos parámetros configurables, inclusive las nuevas clases de planificación asimétricas están parametrizadas.
- Utilizamos diferentes arquitecturas asimétricas, tanto reales como simuladas, sobre las cuales es posible tener varias configuraciones asimétricas.

Desafortunadamente, no existe una herramienta que nos permita administrar todos estos aspectos en conjunto y por lo tanto fue necesario implementarla. Esta herramienta, llamada *Het-harness*, está compuesta por un conjunto de scripts que nos permiten, de una forma más sencilla, establecer la configuración de los escenarios de prueba. Los scripts de *Het-harness* dan soporte a cada uno de los aspectos mencionados tanto para OpenSolaris como para Linux, y pueden incluirse nuevos scripts a medida que van surgiendo nuevos requerimientos. Actualmente, *Het-harness* posee entre todos sus scripts alrededor de 17000 líneas de código.

2.4. Resumen del capítulo y conclusiones

En este capítulo presentamos el entorno experimental utilizado durante la tesis doctoral, compuesto por el hardware asimétrico y el framework de planificación asimétrica.

Durante la primera etapa de la tesis, no existía hardware multicore asimétrico comercial. En esta etapa fue necesario emular sistemas multicore asimétricos mediante la reducción de la frecuencia de algunos cores en CMPs simétricos. A medida que estuvo disponible el hardware AMP real, procedimos a evaluar los algoritmos de planificación usando plataformas de este tipo. Sin embargo, el escaso número de cores que poseen estas plataformas representa una limitación a la hora de evaluar aplicaciones multi-hilo. En este escenario, continuamos emulando el hardware asimétrico sobre plataformas multicore simétricas que poseen mayor número de cores. Por otro lado, las arquitecturas multicore actuales están equipadas con contadores que permiten acceder a medidas de consumo de energía. Estas nuevas arquitecturas nos permitieron realizar experimentos relacionados con la eficiencia energética.

Uno de los desafíos más importantes que presentan los sistemas asimétricos es trasladar sus beneficios a las aplicaciones de usuario. Para evitar modificar las aplicaciones y hacer que el cambio en el hardware resulte transparente al usuario, la responsabilidad recae principalmente en el planificador del sistema operativo. En esta tesis diseñamos un framework de planificación asimétrica y lo implementamos en primer lugar sobre OpenSolaris. Lamentablemente, debido a la falta de mantenimiento de este SO y de soporte para nuevas plataformas hardware, el framework fue portado completamente a Linux, con todo lo que ello implica.

En este capítulo hemos presentado los componentes de nuestro framework, los aspectos fundamentales de los planificadores de OpenSolaris y Linux, y las modificaciones realizadas en cada uno de estos SOs para dar soporte para AMPs.

3 Métricas del planificador y trabajo relacionado

En este capítulo presentamos, en primer lugar, las métricas específicas para cuantificar el grado de alcance de los distintos objetivos (optimización del rendimiento global, la justicia y la eficiencia energética) del planificador en un AMP. A continuación, discutimos los algoritmos de planificación más relevantes propuestos por otros autores en el contexto de los AMPs y ponemos de manifiesto las ventajas y debilidades de cada uno. Asimismo, presentamos las líneas de investigación no abordadas hasta el momento, que son objeto de estudio en esta tesis.

3.1. Métricas del planificador

La mayoría de las métricas existentes para evaluar la efectividad del planificador en distintos escenarios han sido definidas específicamente para sistemas simétricos (SMPs y CMPs convencionales). Lamentablemente, muchas de estas métricas no resultan adecuadas para AMPs. Por lo tanto, para medir la efectividad de las distintas estrategias de planificación propuestas en esta tesis, fue necesario definir nuevas métricas y también adaptar métricas existentes para cuantificar el rendimiento global, la justicia y la eficiencia energética de forma adecuada.

3.1.1. Rendimiento global

Para evaluar el rendimiento global, optamos por descartar el uso de métricas que dependen directamente de las instrucciones por ciclo (IPC) o las instrucciones por segundo (IPS), ya que éstas pueden resultar engañosas a la hora de evaluar el rendimiento de los programas multi-hilo [1]. En particular, en ciertos paradigmas de programación paralela, como OpenMP o Cilk, los hilos pueden realizar esperas activas (*spin*) durante la sincronización con otros hilos. Durante esta espera activa un hilo no realiza ningún trabajo útil y es posible que alcance un IPC o IPS muy alto en la CPU. Este hecho pone de manifiesto que un IPC alto no siempre

implica un rendimiento elevado.

Debido a las limitaciones del IPC, descartamos métricas como *Weighted Speedup* [63] y *Harmonic Mean Speedup* [36]. Para cuantificar el rendimiento global en un AMP de forma más precisa, exploramos distintas métricas que dependen únicamente del tiempo de ejecución de las aplicaciones (*Completion time* o CT). En particular, entre las distintas alternativas analizadas, la métrica que resultó más efectiva es el *Aggregate Speedup* (ASP), que se define como sigue:

$$ASP = \sum_{i=1}^{n} \frac{CT_{Small,i}}{CT_{Sched,i}} - 1 \tag{3.1}$$

donde n es el número de aplicaciones en la carga de trabajo, $CT_{Small,i}$ es el tiempo de ejecución de la aplicación i cuando corre sola en el sistema y utiliza solamente los cores lentos, y $CT_{Sched,i}$ es el tiempo de ejecución de la aplicación i ejecutada bajo un planificador determinado junto al resto de aplicaciones de la carga de trabajo.

Cabe destacar que en el contexto de AMPs, otros autores [12, 64] utilizaron la métrica STP [14] que se define como sigue:

$$STP = \sum_{i=1}^{n} \frac{CT_{alone,i}}{CT_{Sched,i}} \tag{3.2}$$

donde $CT_{alone,i}$ es el tiempo de ejecución de la aplicación i cuando se ejecuta sola en el sistema. En un sistema asimétrico $CT_{alone,i}$ se minimiza usando todos los cores rápidos disponibles.

Durante estudios preliminares detectamos que el ASP (métrica propuesta en esta tesis) captura de forma más precisa que el STP las diferencias en el rendimiento global que proporcionan distintos algoritmos de planificación. Además, observamos que el STP, a diferencia del ASP, no permite cuantificar el beneficio global que una carga de trabajo obtiene al usar los cores rápidos de la plataforma. Este hecho provoca que dos cargas de trabajo con muy distinta composición puedan tener asociado el mismo valor de STP, a pesar de que una de las cargas utilice más eficientemente los cores rápidos que la otra.

Para mostrar la limitación de STP, supongamos un sistema asimétrico formado por un core rápido y uno lento. Sobre este sistema se ejecutan 2 cargas de trabajo (A y B), ejecutadas una detrás de la otra. Cada carga está formada por dos aplicaciones secuenciales con las siguientes características:

- **Carga A**: Las dos aplicaciones de la carga tardan el mismo tiempo al ejecutarse en un core rápido que en uno lento.
- **Carga B**: La primera aplicación de la carga se ejecuta dos veces más rápido en un core rápido que en uno lento; la segunda aplicación no experimenta ningún beneficio al ejecutarse en cores rápidos.

Suponiendo despreciable el impacto de la contención por recursos compartidos en el AMP y teniendo en cuenta que el planificador del SO siempre asigna la primera aplicación de la carga de trabajo a un core rápido y la segunda la asigna a un core lento, el STP valdrá 2 para ambas cargas de trabajo ($CT_{alone} = CT_{Sched}$ para las dos aplicaciones en ambos escenarios). De este modo, el STP no refleja el hecho de que la carga *B* obtiene beneficios del core rápido (primera aplicación) y no así la *A*. De hecho, la carga *A* experimentaría el mismo rendimiento si el core rápido del AMP se reemplazara por uno lento. Por el contrario, la métrica ASP sí es capaz de capturar el grado de utilización de los cores rápidos por las aplicaciones de la carga de trabajo. De hecho, para la carga *A* el ASP asociado es 0: la carga no obtiene ningún beneficio extra al utilizar cores rápidos con respecto a los cores lentos. Sin embargo, para la segunda carga, ASP=1 lo que significa que esta carga sí obtiene beneficios del core rápido.

3.1.2. Justicia

En el contexto de la planificación de procesos se han empleado dos definiciones muy diferentes de justicia. En la primera definición, se considera que un algoritmo de planificación es justo si es capaz de asignar la misma porción de tiempo de CPU a aplicaciones con la misma prioridad [34, 33]. La segunda definición de justicia establece que un algoritmo de planificación es justo si garantiza que las aplicaciones de una carga de trabajo con la misma prioridad experimentan la misma degradación del rendimiento (*slowdown*) al ejecutarse de forma simultánea con otras aplicaciones; esta degradación se mide con respecto a cuando cada aplicación se ejecuta sola en el sistema. La segunda definición de justicia se utiliza más ampliamente en el contexto de los CMPs en la actualidad [16, 42], porque esta aproximación permite cuantificar la degradación en rendimiento que las aplicaciones pueden experimentar al competir por el uso de recursos compartidos en un CMP (por ejemplo, niveles de cache o ancho de banda con memoria). Por este motivo, en esta tesis empleamos esta noción de justicia, más adecuada para sistemas multicore. Para cuantificar la justicia, trabajos previos han empleado la *injusticia* o *unfairness* [42, 13], una métrica que se define como:

$$Unfairness = \frac{max(Slowdown_1, Slowdown_2, \cdots, Slowdown_n)}{min(Slowdown_1, Slowdown_2, \cdots, Slowdown_n)} \tag{3.3}$$

Donde *Slowdown* para una aplicación i se define como sigue:

$$Slowdown_i = \frac{CT_{Sched,i}}{CT_{Fast,i}} \tag{3.4}$$

$CT_{Fast,i}$ es el tiempo de ejecución de la aplicación i cuando corre sola en el sistema (con todos los cores rápidos disponibles para ella). Es importante mencionar, que adaptamos esta definición de *unfairness* para sistemas AMPs, ya que en la definición original [42] para CMPs utiliza $CT_{alone,i}$ en lugar de $CT_{Fast,i}$ en la ecuación 3.4.

3.1.3. Eficiencia energética

Para cuantificar la eficiencia energética asociada a la ejecución de una carga de trabajo en un AMP bajo un algoritmo de planificación determinado, utilizamos la métrica*Energy-Delay Product* (EDP)[24, 18], que se define como:

$$EDP = \frac{Energia\_total\_consumida \cdot CT}{Instrucciones\_retiradas\_totales} \tag{3.5}$$

donde *CT* es el tiempo de ejecución de la carga de trabajo (tiempo de ejecución de la aplicación más lenta).

Si bien el ASP es una métrica a maximizar, cuanto más reducido sea el valor del *EDP* y del *unfairness*, mejor será la eficiencia energética y el grado de justicia proporcionada, respectivamente.

3.2. Trabajo relacionado

Distintos autores han puesto de manifiesto los beneficios de los AMPs sobre los sistemas multicore simétricos convencionales [30, 3, 23, 41, 64, 20]. En el *survey* elaborado por Sparsh Mittal puede encontrarse un resumen de las principales conclusiones de estos autores [40]. A pesar de los beneficios de los AMPs, estas arquitecturas plantean desafíos importantes para el software de sistema [17, 50, 10, 40]. La planificación de procesos a nivel de sistema operativo es uno de los desafíos más significativos [17, 40], y éste es el aspecto central de esta tesis doctoral.

Antes del inicio de la tesis, la mayoría de los algoritmos de planificación existentes para AMPs estaban orientados a optimizar el rendimiento global del sistema [30, 8, 60, 54, 58]. Sin embargo, hasta la fecha, la optimización de otros aspectos críticos como la justicia o el consumo de energía, no habían recibido suficiente atención por parte de la comunidad científica. Asimismo, ninguno de los algoritmos propuestos hasta el momento tenía la capacidad de ofrecer un soporte robusto de prioridades. El principal objetivo de esta tesis doctoral fue llenar este vacío, mediante el diseño de estrategias de planificación más versátiles.

Durante el desarrollo de la tesis, otros investigadores propusieron distintos algoritmos de planificación para mejorar la justicia, el rendimiento global y la eficiencia energética en AMPs. Sin embargo, ninguno de estos trabajos analiza la interrelación que existe entre estos tres objetivos. En esta tesis doctoral, cubrimos esta limitación, mediante la realización de distintos análisis teóricos y experimentales exhaustivos. Los resultados obtenidos revelan que los planificadores que intentan optimizar una métrica específica (rendimiento global o eficiencia energética) pueden llegar a degradar otras métricas de forma significativa en AMPs. Esta tesis también muestra las capacidades de distintos algoritmos de planificación a la hora optimizar las distintas métricas en hardware multicore asimétrico real. Las conclusiones de los distintos análisis teóricos y experimentales realizados han sido clave para diseñar el

algoritmo ACFS-E, la propuesta más versátil de esta tesis doctoral. Este algoritmo, que se describe en el capítulo 6, constituye el primer planificador para AMPs que combina en un único algoritmo la capacidad de optimizar justicia, rendimiento, y eficiencia energética.

El resto de esta sección está estructurado en tres partes. En primer lugar presentamos las propuestas orientadas a optimizar el rendimiento en un AMP. A continuación, describimos las principales estrategias para proporcionar justicia en multicore asimétricos. Finalmente, presentamos una descripción de distintas técnicas orientadas a reducir el consumo de energía en un AMP.

3.2.1. Optimización del rendimiento

Para maximizar el rendimiento del sistema en el contexto de cargas de trabajo multiprogramadas, estudios previos han demostrado que el planificador debe ejecutar en cores rápidos aquellas aplicaciones que obtienen un mayor beneficio relativo (*speedup*) al usar este tipo de cores [30, 8]. Por simplicidad, nos referiremos a esta aproximación como HSP (*High SPeedup*). Para poder implementar este tipo de estrategias, el SO necesita un mecanismo para poder determinar en tiempo de ejecución el factor de ganancia o *SF* (*Speedup Factor*) de cada hilo que se encuentra en ejecución. El *SF* de un hilo se define como sigue:

$$SF = \frac{IPS_{fast}}{IPS_{slow}} \tag{3.6}$$

donde IPS_{fast} e IPS_{slow} son los ratios de instrucciones por segundo obtenidos por el hilo en un core rápido y un core lento respectivamente. Para las aplicaciones puramente secuenciales el *SF* del único hilo activo representa el beneficio relativo (*speedup*) que la aplicación experimenta al ejecutarse en un core rápido con respecto a uno lento.

Es importante mencionar que HSP es un algoritmo inherentemente injusto, ya que no hace ningún esfuerzo por compartir los cores rápidos entre las aplicaciones o hilos. HSP realiza la asignación de hilos a core en función del *speedup* que éstas alcanzan en el AMP. Los cores rápidos se asignan a la aplicación con el *speedup* más alto. Si la aplicación es incapaz de utilizar todos los cores rápidos del AMP (la cantidad de hilos ejecutables es menor que el número de cores rápidos), los cores rápidos restantes se asignan a la aplicación con el segundo *speedup* más alto, y así sucesivamente.

3.2.1.1. Técnicas para determinar el SF en tiempo de ejecución

La principal diferencia entre las distintas variantes de la estrategia HSP [30, 8, 60, 54, 28, 59] se encuentra en el mecanismo utilizado para obtener el *speedup factor* de un hilo en tiempo de ejecución. Para esto, se han propuesto varias técnicas que se pueden agrupar en tres grandes categorías:

- Medición directa o *IPC Sampling* [30, 8, 64].
- Técnicas basadas en modelos de estimación [28, 58], que aproximan el SF de forma indirecta a partir de distintas métricas de rendimiento monitorizadas en tiempo de ejecución.
- Técnicas asistidas por hardware específico [12].

El primer enfoque, que mide el *speedup factor* directamente [30, 8], implica monitorizar continuamente el rendimiento del hilo (IPC) cuando se ejecuta en ambos tipos de core. El SF actual del hilo se determina usando los valores de IPC obtenidos en ambos tipos de cores durante el último intervalo de muestreo. Trabajos anteriores han demostrado que este enfoque, conocido como *IPC Sampling*, está sujeto a imprecisiones en el cálculo de *speedup factor* asociadas a los cambios de fase de las aplicaciones [60, 56]. Los resultados obtenidos en esta tesis doctoral nos han permitido corroborar esta limitación del*IPC Sampling*, durante la evaluación experimental de algoritmos basados en esta técnica [64].

El segundo enfoque estima en tiempo de ejecución el *speedup factor* de un hilo basándose en el valor de distintas métricas de rendimiento (tasa de fallos de cache, IPC, tasa de predicción de saltos,...) obtenidas únicamente en el core donde el hilo se ejecuta actualmente [28, 58, 47]. Para obtener los valores de estas métricas, el planificador usa los contadores hardware del procesador en tiempo de ejecución. Investigaciones previas [60, 59] demostraron que los planificadores que se basan en modelos de estimación de *speedup factor* pueden obtener valores más precisos que empleando medición directa. Asimismo, reducen el *overhead* de la medición directa, que requiere migrar hilos periódicamente para monitorizar su rendimiento en ambos tipos de core[8].

La propuesta más destacada que sigue el tercer enfoque es PIE (*Performance Impact Estimation*), un mecanismo asistido por hardware para aproximar el SF de forma precisa [12]. Lamentablemente, el soporte hardware requerido por PIE, que hasta la fecha fue evaluado sólo en simuladores de arquitecturas, presenta ciertas limitaciones que hacen difícil su integración en sistemas multicore asimétricos reales [48].

Debido a los problemas prácticos asociados a PIE y a la estrategia de medición directa del SF, los algoritmos propuestos en esta tesis doctoral utilizan el enfoque basado en modelos de estimación de *speedup factor*. Es importante destacar que la estimación del *speedup factor* requiere derivar modelos de rendimiento específicamente adaptados a la plataforma y al tipo de core utilizado. En el capítulo 4 utilizamos la metodología propuesta en [53] para la construcción de modelos de estimación de *speedup factor*. Esta técnica permite obtener modelos de estimación precisos y sencillos para AMPs donde los cores poseen la misma microarquitectura y la misma jerarquía cache. Un ejemplo de esto son los sistemas asimétricos donde los cores difieren sólo en la frecuencia del procesador o en la tasa de instrucciones retiradas [53]. Sin embargo, en sistemas AMP donde los cores tienen diferencias más profundas a nivel microarquitectónico, como en el prototipo *QuickIA* de Intel, esta técnica no permite

obtener modelos de estimación de SF precisos. Por esta razón, fue necesario desarrollar una nueva metodología para obtener modelos robustos en escenarios más complejos, como el prototipo QuickIA. En el capítulo 6 trataremos en detalle este tema.

3.2.1.2. Soporte para aplicaciones multi-hilo

Trabajos recientes han demostrado que cuando la carga de trabajo incluye aplicaciones multi-hilo, tomar decisiones de planificación basadas únicamente en el SF de cada hilo puede llevar al planificador a degradar el rendimiento global de forma significativa [3, 54, 58, 53]. Esto se debe a que el SF de los hilos de una aplicación paralela no siempre permite aproximar el beneficio que la aplicación como un todo obtiene al utilizar los cores rápidos en un AMP [23]. Por esta razón, además del SF hay que tener en cuenta otros aspectos, como por ejemplo la cantidad de paralelismo a nivel de hilo (TLP) presente en la aplicación [3, 23]. La clave en estos escenarios está en determinar el beneficio global (*speedup*) que la aplicación multi-hilo como un todo obtiene al utilizar los escasos cores rápidos de la plataforma, con respecto a usar sólo los cores lentos. En particular, las aplicaciones paralelas escalables con un gran número de hilos experimentan un *speedup* bajo al usar los cores rápidos de un AMP, debido a que en la práctica no hay cores rápidos disponibles para todos los hilos, y esto limita el rendimiento de la aplicación. Por este motivo, en cargas de trabajo que combinan aplicaciones paralelas con secuenciales, asignar preferente las aplicaciones secuenciales a los cores rápidos permite obtener un mayor rendimiento global [54, 58].

En [58, 53] se derivan analíticamente distintas fórmulas para aproximar el *speedup* de aplicaciones paralelas regulares (por ejemplo OpenMP). En esta tesis (sección 4.2.2), derivamos fórmulas adicionales para aproximar el *speedup* de otros tipos de aplicaciones multi-hilo. Estas fórmulas dependen del número de hilos activos de la aplicación (que aproximan el grado de TLP), el SF de los hilos y el número de cores rápidos en el AMP. Los algoritmos Prop-SP y ACFS, propuestos en esta tesis, emplean estas fórmulas en tiempo de ejecución para aproximar el *speedup* de las distintas aplicaciones.

Otros autores han propuesto soporte específico para acelerar la ejecución de aplicaciones multi-hilo sobre AMPs [3, 32, 54, 26, 27, 39, 25]. La mayoría de estas propuestas utilizan los cores rápidos para acelerar fases secuenciales de ejecución y otros cuellos de botella presentes en las aplicaciones paralelas, empleando distintas técnicas software [3, 54] o hardware [26, 27, 39]. Algunas de estas propuestas explotan la interacción del SO con el*runtime system*, que se ejecuta en modo usuario [32, 54].

En [26] los autores proponen el algoritmo de planificación BIS, un mecanismo hardware-software para identificar y acelerar un amplio espectro de cuellos de botella. BIS identifica los cuellos de botella más críticos en una aplicación teniendo en cuenta el número de ciclos que los hilos han esperado en cada uno de ellos, y acelera estos cuellos de botella utilizando los cores rápidos de un AMP. Más recientemente, los mismos autores propusieron UBA [27], un mecanismo que extiende BIS e incluye soporte para la aceleración de hilos rezagados (*lagging*

threads) es decir, hilos que tardan más en ejecutar que otros a causa del desbalance de carga u otros aspectos microarquitectónicos como por ejemplo fallos de caché. En [39], los autores proponen el algoritmo de planificación *Thread Lock Section Scheduler* (TLSS), una técnica de hardware que identifica cuellos de botella de aplicaciones multi-hilo. En comparación con BIS y UBA, TLSS no requiere extensiones ISA y requiere muchas menos extensiones del hardware. Sin embargo, no es capaz de identificar todas las secciones críticas en una aplicación. Si bien la mayoría de las técnicas anteriormente citadas están enfocadas a aplicaciones HPC, en [25] se proponen mecanismos para proporcionar soporte para aplicaciones multi-hilo irregulares y con escalabilidad limitada, que están presentes en entornos de escritorio. Cabe destacar que las propuestas de planificación a nivel de sistema operativo en esta tesis son ampliamente ortogonales a estas estrategias hardware/software.

3.2.2. Justicia y soporte a prioridades

La primera estrategia de planificación orientada a proporcionar justicia en AMPs fue el algoritmo *asymmetry-aware round-robin* [8]. Por simplicidad, utilizaremos el nombre "RR" para hacer referencia a este algoritmo. Esta estrategia de planificación distribuye de forma equitativa los ciclos de core rápido entre aplicaciones, llevando a cabo migraciones periódicas de hilos, pero sin tener en cuenta el *speedup* ni las prioridades de usuario. RR puede implementarse de forma sencilla en la mayoría de sistemas operativos de propósito general, y no requiere soporte hardware[1]. RR ha sido ampliamente usado como base para la comparación con otros algoritmos de planificación [8, 54, 53].

En esta tesis (capítulo 5), demostramos que RR es una estrategia de planificación subóptima, tanto desde el punto de vista de la justicia como del rendimiento [52, 57]. A pesar de esto, se ha demostrado que la distribución equitativa de los ciclos de core rápido entre aplicación que realiza RR brinda un mayor rendimiento en AMP que el alcanzado por los algoritmos de planificación por defecto de los sistemas operativos de propósito general, que en su mayoría no son conscientes de la asimetría en la plataforma. Asimismo, al contrario que estos algoritmos, RR proporciona tiempos de ejecución repetibles en distintas ejecuciones de una misma carga de trabajo[33]. Es importante mencionar, que éste no es el caso del algoritmo de planificación por defecto del kernel Linux (CFS), ni tampoco de la variante de este planificador creada para procesadores ARM big.LITTLE (parche HMP[37]). Esto se debe a que estas dos estrategias del kernel Linux pueden asignar una aplicación a diferentes tipos de cores en ejecuciones sucesivas de la misma carga de trabajo. Esto provoca una enorme variación en los tiempos de ejecución de una aplicación, lo que provoca una alta variabilidad de los valores de las métricas presentadas en la sección 3.1. Específicamente, en nuestro entorno experimental observamos que el tiempo de ejecución de una aplicación bajo el planificador por defecto de Linux puede incrementarse hasta 4.7 veces con respecto a la ejecución más rápida registrada. Para evitar sacar conclusiones erróneas de nuestros experimentos, en esta tesis no mostramos los resultados bajo el planificador por defecto de Linux (CFS, con o sin el parche HMP).

[1] En [38] se propone una implementación hardware de este algoritmo.

El algoritmo de planificación A-DWRR propuesto en [33] tiene como objetivo proporcionar justicia y soporte a prioridades en sistemas AMP. Para llevar a cabo este objetivo, A-DWRR tiene en cuenta la potencia de cómputo de los diferentes tipos cores al realizar el *CPU accounting* para cada hilo. Concretamente, este algoritmo emplea un concepto especial de tiempo de CPU, extendido para AMPs, conocido como *tiempo de CPU escalado*. Utilizando este concepto, los ciclos de CPU consumidos en un core rápido tienen mayor peso que los ciclos consumidos en un core lento. Para garantizar justicia, A-DWRR trata de equilibrar el tiempo de CPU escalado que consumen los distintos hilos que se encuentran en ejecución, considerando también la prioridad de cada uno de ellos. Un aspecto negativo de A-DWRR, es el hecho de que no tiene en cuenta que las aplicaciones puedan exhibir distintos *speedups* al utilizar los cores rápidos y que el *speedup* puede variar a lo largo del tiempo. Por otra parte, A-DWRR realiza un reparto del tiempo de CPU de forma equilibrada entre todos los hilos del sistema, en lugar hacer este reparto entre aplicaciones, lo cual favorece a las aplicaciones con mayor TLP. Como revelan nuestros resultados experimentales, este reparto puede originar una degradación significativa del rendimiento del sistema, e impide a A-DWRR garantizar que las aplicaciones con la misma prioridad sufran una degradación en rendimiento (*slowdown*) similar al compartir el sistema. En particular, la degradación es especialmente significativa cuando la carga de trabajo incluye aplicaciones multi-hilo.

El algoritmo Prop-SP [52, 57] propuesto en esta tesis, constituye la primera aproximación de planificación en AMPs que explota la diversidad de *speedups* presente en una carga de trabajo para mejorar la justicia y el soporte a prioridades en AMPs. Prop-SP intenta equiparar la degradación que experimentan las aplicaciones con igual prioridad manteniendo el rendimiento del sistema en valores aceptables. Para hacer esto posible, cada aplicación recibe una fracción de tiempo en los cores rápidos proporcional al producto entre su *speedup* y su prioridad. El capítulo 4 presenta el algoritmo de planificación Prop-SP en detalle. El análisis experimental de ese capítulo muestra que Prop-SP proporciona un mejor compromiso rendimiento-justicia y un soporte de prioridades más robusto que A-DWRR, la estrategia de planificación orientada a justicia de referencia hasta la fecha. A pesar de esto, Prop-SP no es un algoritmo que optimiza la justicia, como demostramos en el capítulo 5.

Poco después de la creación del algoritmo Prop-SP, Van Craeynest y otros presentaron el algoritmo de planificación *Equal-progress* [64]. Por simplicidad, nos referiremos al algoritmo *Equal-progress* como EQP. Al igual que Prop-SP, EQP explota la diversidad de SFs entre hilos de la carga de trabajo para mejorar la justicia en multicore asimétricos. EQP es la propuesta que presenta una mayor similitud con el algoritmo ACFS [56], propuesto en esta tesis doctoral. En particular, tanto EQP como ACFS fueron diseñados para optimizar la justicia en AMPs. Para lograr este objetivo, ambos algoritmos monitorizan el progreso realizado por cada hilo en un AMP, e intentan ofrecer justicia garantizando un equilibrio entre el progreso de los distintos hilos, lo cual requiere migrar hilos entre cores cada cierto tiempo. A pesar de las similitudes entre ACFS y EQP, existen diferencias importantes entre ambas propuestas. En primer lugar, a la hora de cuantificar el progreso de un hilo, EQP no considera las distintas fases de SF que el hilo atraviesa durante su ejecución. Específicamente, EQP aproxima este progreso

considerando sólo el número total de ciclos que el hilo ha consumido hasta el momento en cada tipo de core y su SF actual. Por el contrario, ACFS mantiene contadores de progreso por hilo, que se actualizan en cada *tick* de reloj en base al *speedup* que la aplicación a la que el hilo pertenece extraería *en ese momento* si utilizara todos los cores disponibles en el AMP. En segundo lugar, para determinar el SF de un hilo en tiempo de ejecución EQP se basa en *IPC-sampling* o PIE[2], mientras que ACFS emplea modelos de estimación basados en contadores hardware. En tercer lugar, EQP intenta ofrecer justicia entre aplicaciones garantizando que cada hilo de la carga realiza el mismo progreso (basado únicamente en el SF). Sin embargo, esto no garantiza una degradación del rendimiento uniforme entre aplicaciones en el caso de que la carga de trabajo incluya aplicaciones multi-hilo. ACFS resuelve este problema considerando el *speedup* de la aplicación como un todo a la hora de actualizar los contadores de progreso de cada hilo. Finalmente, EQP no ofrece soporte a prioridades, mientras que ACFS sí está provisto de este soporte. Describiremos en detalle ACFS y sus beneficios en el capítulo 6.

3.2.3. Optimizando la eficiencia energética

Otros investigadores han propuesto formas para reducir el consumo de energía en AMP [41, 31, 45, 44, 46, 65]. Mogul y otros proponen el uso de cores lentos de un AMP para ejecutar llamadas al sistema[41]. En particular, los autores modifican el sistema operativo para forzar la ejecución de una llamada al sistema en un core de bajo consumo. Esta estrategia se basa en la observación de que las llamadas al sistema y el código del sistema operativo en general usan los cores rápidos de forma ineficiente. Siguiendo una aproximación similar, Kumar y Fedorova [31] proponen asignar a cores lentos el dominio de control *dom0* del hipervisor *Xen*. Estas modificaciones del software de sistema [41, 31] son ortogonales a las propuestas realizadas en esta tesis doctoral.

Petrucci en [44] propone *Octopus-Man*, una estrategia que tiene como objetivo garantizar calidad de servicio en cargas de trabajo formadas por tareas sensibles a la latencia (por ejemplo, servidores web). Al mismo tiempo, esta estrategia intenta maximizar la eficiencia energética de todo el sistema. Cabe destacar que éste es uno de los pocos trabajos que emplea el prototipo *QuickIA* de Intel [10], uno de los sistemas asimétricos reales que utilizamos para llevar a cabo nuestros experimentos. Sin embargo, los autores no proponen un mecanismo para obtener estimaciones precisas de SF por hilo para este prototipo. Obtener un modelo robusto de estimación de *SF* sobre QuickIA fue uno de los grandes desafíos de esta tesis doctoral. La metodología propuesta en esta tesis (descrita en el capítulo 6), nos permitió superar este desafío.

Más recientemente, Petrucci y otros [46] proponen una estrategia de planificación en modo usuario cuyo objetivo es optimizar la eficiencia energética en AMPs. Esta estrategia

[2] En nuestra evaluación experimental sobre hardware asimétrico real, estudiamos la variante de EQP basada en IPC sampling. Como se mencionó anteriormente, PIE no está disponible en ningún AMP real, y las limitaciones del soporte hardware asociado [48] complican la implementación de PIE en sistemas asimétricos comerciales.

está basada en un modelo de programación lineal entera (ILP) y emplea un *solver* específico basado en el modelo para determinar la asignación de hilos a cores más prometedora. Cabe destacar que la estrategia presentada en [46] está específicamente diseñada para sistemas empotrados. En este escenario, las tareas de la carga de trabajo exhiben patrones de ejecución más predecibles que en sistemas de propósito general, por lo que en este contexto es posible realizar optimizaciones adicionales. Por el contrario, las estrategias de planificación propuestas en esta tesis para mejorar la eficiencia energética (algoritmos EEF-Driven y ACFS-E) están pensadas para sistemas de propósito general, y no se basan en ninguna suposición sobre las características de la carga de trabajo. Otra diferencia con respecto a nuestro trabajo es el hecho de que Petrucci evalúa su propuesta usando multicore simétricos emulados (cores con distinta frecuencia) en lugar de usar hardware AMP real. En este entorno experimental simplificado, los autores emplean modelos de estimación para aproximar el SF de un hilo en tiempo de ejecución usando un modelo de regresión que requiere la monitorización de las métricas MIPS y fallos de cache de último nivel del hilo en el core actual. Como demostramos en esta tesis, en sistemas AMP reales, donde cores rápidos y lentos pueden presentar características muy diversas (por ejemplo, distinta microarquitectura o jerarquía cache), usar sólo estas dos métricas de rendimiento no permite aproximar el SF de forma precisa mediante regresión.

En [65] los autores proponen PRIM, un algoritmo de planificación guiado por reglas cuyo objetivo es reducir el consumo de energía en AMPs. A grandes rasgos, este algoritmo funciona de la siguiente manera. Inicialmente, cuando un nuevo hilo entra al sistema, el planificador lo asigna a un core del tipo de manera de mantener balanceada la carga entre todos los cores. Periódicamente, PRIM selecciona aleatoriamente un par de hilos: uno de ellos que esté asignado actualmente a un core rápido (T_a), y el otro asignado a un core lento (T_b). A continuación, el algoritmo estima, de acuerdo a determinadas reglas[3] específicas de plataforma, si el intercambio de estos hilos permite un ahorro de energía. Si esto es así, el planificador realiza el intercambio: el hilo T_a pasará a ejecutarse en el core lento y el hilo T_b pasará a ejecutarse en el core rápido. En la propuesta original de PRIM [65], los autores evaluaron la efectividad de este algoritmo sobre un entorno de simulación. Por este motivo, para poder comparar PRIM con nuestras propuestas de planificación, sobre una plataforma asimétrica real, fue necesario crear una implementación de este algoritmo el kernel Linux. Esto implicó adaptar las reglas presentadas en [65] a la plataforma asimétrica utilizada. Nuestros resultados experimentales revelan que PRIM está sujeto a dos limitaciones importantes. En primer lugar, el algoritmo no siempre consigue optimizar el consumo de energía y además en ciertos escenarios degrada el rendimiento global de forma significativa. Por otro lado, la naturaleza de las reglas en las que se basa PRIM, llevan al planificador a efectuar asignaciones de hilos a cores subóptimas. Cabe destacar que éstas reglas no cuantifican el ahorro energético resultante de realizar un intercambio de hilos, sino que indican únicamente si ese intercambio es beneficioso o no en términos de consumo de energía. Debido a esta limitación, PRIM no es capaz de detectar si existe un hilo T_c corriendo en un core rápido tal que un intercambio

[3] Evaluar estas reglas específicas de plataforma requiere que el SO monitorice en tiempo de ejecución distintas métricas de rendimento de los hilos (como el IPC o la tasa de fallos de último nivel de cache) usando contadores hardware.

con este hilo con T_b resulta en un ahorro de energía mayor que el intercambio entre T_a y T_b. Nuestro análisis experimental demuestra que el algoritmo EEF-Driven, descrito en el capítulo 7, supera las limitaciones de PRIM, ofreciendo mayor eficiencia energética para un amplio espectro de cargas de trabajo.

3.3. Resumen del capítulo y conclusiones

En este capítulo definimos las métricas empleadas en la tesis para cuantificar la efectividad de las distintas estrategias de planificación propuestas, y además proporcionamos una visión general de los algoritmos de planificación más relevantes para AMPs propuestos por otros autores.

En la evaluación experimental que realizamos en esta tesis doctoral empleamos las métricas *Aggregate Speedup* (*ASP*), *Unfairness* y *Energy-Delay Product* (EDP), que permiten medir el rendimiento global, la justicia y la eficiencia energética, respectivamente. Al inicio de la tesis llevamos a cabo un estudio para seleccionar y definir las métricas adecuadas para nuestra evaluación, ya que la mayoría de las métricas existentes, definidas específicamente para sistemas simétricos, no resultaban efectivas para AMPs.

Hasta el comienzo de esta tesis doctoral, la mayoría de los algoritmos de planificación para AMPs estaban orientados a optimizar el rendimiento global del sistema. Sin embargo, la optimización de otros aspectos como la justicia o el consumo de energía, no habían recibido suficiente atención por parte de la comunidad científica. Asimismo, ninguno de los algoritmos propuestos hasta el momento tenía la capacidad de ofrecer un soporte robusto de prioridades de usuario. El principal objetivo de esta tesis doctoral fue llenar este vacío, mediante el diseño de estrategias de planificación más versátiles.

Para maximizar el rendimiento del sistema en el contexto de cargas de trabajo multiprogramadas, estudios previos han demostrado que el planificador debe seguir el enfoque HSP (*High SPeedup*) [30, 8], es decir, ejecutar en cores rápidos aquellas aplicaciones que obtienen un mayor beneficio relativo (*speedup*) al usar este tipo de cores [30, 8]. Para implementar este tipo de estrategias, el SO necesita un mecanismo para determinar en tiempo de ejecución el *speedup factor* (SF) de cada hilo que está corriendo en el sistema.

Existen al menos tres mecanismos: IPC-Sampling, técnicas asistidas por hardware específico y mediante modelos de estimación de SF. Debido a las limitaciones de las dos primeras técnicas, en esta tesis doctoral utilizamos el tercer enfoque. La estimación de SF requiere derivar modelos de rendimiento específicamente adaptados a la plataforma y al tipo de core utilizado.

Cuando la carga de trabajo incluye aplicaciones multi-hilo, tomar decisiones de planificación basadas únicamente en el SF de cada hilo puede provocar una degradación sustancial del rendimiento global del sistema. Por esta razón, en esta tesis doctoral, derivamos fórmulas

adicionales para aproximar el *speedup* de la aplicación multi-hilo como un todo. Estas fórmulas tienen en cuenta el número de hilos activos de la aplicación (que aproximan el grado de TLP), el *SF* de sus hilos y el número de cores rápidos en el AMP.

La primera estrategia de planificación orientada a proporcionar justicia en AMPs fue *asymmetry-aware round-robin* o RR [8]. Posterior a RR, en [33] los autores propusieron A-DWRR, que tiene como objetivo garantizar justicia y proporcionar soporte a prioridades en sistemas AMP. Un aspecto negativo de estos dos algoritmos es que no tienen en cuenta que las aplicaciones pueden exhibir distintos *speedups* al utilizar los cores rápidos y que el *speedup* puede variar a lo largo del tiempo. En esta tesis proponemos Prop-SP [52, 57] (capítulo 4), que constituye la primera aproximación de planificación en AMPs que explota la diversidad de *speedups* presente en una carga de trabajo para mejorar la justicia y el soporte a prioridades.

Después de la creación de Prop-SP, otros autores propusieron el algoritmo de planificación *Equal-progress* o EQP [64]. Al igual que Prop-SP, EQP explota la diversidad de *SF*s entre hilos de la carga de trabajo para garantizar justicia en AMPs. En el capítulo 6 proponemos el algoritmo de planificación orientado a justicia ACFS [56], que supera a EQP. ACFS surge del estudio analítico presentado en el capítulo 5, donde desarrollamos un modelo teórico para aproximar el planificador óptimo de justicia.

Para diseñar estrategias orientadas a optimizar la eficiencia energética, extendimos el modelo teórico mencionado anteriormente con la capacidad de aproximar el *EDP*. El nuevo modelo nos permitió aproximar el planificador teórico que optimiza el *EDP*, que fue clave para la implementación de nuestras propuestas de planificación EEF-Driven y ACFS-E. En nuestro análisis experimental del capítulo 7 mostramos que EEF-Driven supera a PRIM [65], un algoritmo de planificación guiado por reglas cuyo objetivo es reducir el consumo de energía en AMPs.

4 Mejorando el compromiso rendimiento-justicia

En este capítulo describimos el algoritmo de planificación Prop-SP, la primera propuesta de planificación de esta tesis doctoral. El objetivo de Prop-SP es ofrecer un buen equilibrio entre justicia y rendimiento global. Para lograr este objetivo, Prop-SP está provisto de un mecanismo para determinar en tiempo de ejecución el beneficio relativo (*speedup*) que una aplicación alcanza al usar los cores rápidos con respecto a usar los cores lentos en un AMP. Asimismo, este algoritmo tiene en cuenta las prioridades definidas por el usuario a la hora de tomar decisiones de planificación, y además proporciona soporte específico para aplicaciones multi-hilo.

Comenzamos este capítulo con la descripción del planificador Prop-SP y el soporte que ofrece para aplicaciones multi-hilo. A continuación presentamos la estrategia que emplea Prop-SP para estimar el *speedup* de una aplicación en tiempo de ejecución. Para finalizar, realizamos un análisis experimental donde comparamos Prop-SP con algunos de los algoritmos del estado del arte para AMP. Asimismo, mostramos la efectividad del algoritmo Prop-SP cuando se tiene en cuenta las prioridades de usuario.

4.1. El algoritmo de planificación Prop-SP

El algoritmo de planificación Prop-SP asigna hilos a cores rápidos y cores lentos intentando preservar el balance de carga en el sistema, y cada cierto tiempo migra hilos entre los distintos tipos de cores para asegurar que puedan ejecutarse en los cores rápidos por un período de tiempo específico. Para distribuir eficientemente los ciclos de core rápido entre las aplicaciones, Prop-SP emplea dos mecanismos fundamentales: la distribución proporcional de ciclos de core rápido entre las aplicaciones (en base a sus prioridades) y el intercambio periódico de hilos entre distintos tipos de cores. Para garantizar que una aplicación recibe una fracción específica de tiempo en los cores rápidos Prop-SP utiliza una estrategia basada en créditos que está inspirada en la que sigue el planificador Credit Scheduler de Xen [9] sobre multicores simétricos.

4.1.1. Asignación de créditos de cores rápidos

La asignación de créditos de cores rápidos es un mecanismo para controlar la cantidad de ciclos consumidos por los hilos que están en ejecución sobre los cores rápidos de un AMP. Este mecanismo funciona del siguiente modo. Cada hilo tiene asociado un contador de *créditos* de core rápido. Aquellos hilos que poseen *créditos*, es decir tienen su contador mayor a cero, pueden ejecutarse en este tipo de cores. Cuando un hilo se ejecuta en un core rápido, sus créditos se van consumiendo: el contador de créditos asociado se decrementa.

Cada cierto tiempo, Prop-SP inicia un proceso de asignación de créditos que otorga créditos de core rápido a las aplicaciones activas (aquellas que poseen hilos listos para ejecutar). Llamamos *período de ejecución* al período de tiempo que transcurre entre dos períodos consecutivos de asignación de créditos. El algoritmo de planificación determina este período dinámicamente.

La idea de asignación de créditos está basada en el planificador *Xen's Credit Scheduler* (CS) [9]. Sin embargo, la distribución de créditos que lleva a cabo Prop-SP es completamente diferente a la que realiza CS. Mientras CS usa los créditos para distribuir el tiempo de ejecución de CPUs virtuales entre CPUs físicas de un sistema, Prop-SP utiliza créditos para distribuir el tiempo de uso de cores rápidos entre aplicaciones en un sistema AMP.

Para asignar créditos a una aplicación, Prop-SP tiene en cuenta el peso dinámico, que se define como el producto de su *speedup neto* (*speedup* -1) y su peso estático. En este contexto, el *speedup* representa el beneficio relativo que la aplicación debería alcanzar si todos los cores rápidos en el AMP están dedicados a ejecutar hilos de esta aplicación, con respecto a correr todos los hilos en los cores lentos. Prop-SP estima el *speedup* en tiempo de ejecución utilizando un modelo de estimación específico de la plataforma, describiremos esto en detalle en la sección 4.2.2. El peso estático de una aplicación se obtiene a partir de su prioridad, establecida por el usuario.

El proceso de asignación de créditos, se describe en el algoritmo 4.1. Este algoritmo consta de tres pasos:

1. Calcular los pesos dinámicos de cada aplicación y la suma de los pesos dinámicos de todas las aplicaciones.

2. Asignar créditos a cada aplicación en función de su peso dinámico y de la suma de los pesos dinámicos de todas las aplicaciones.

3. Calcular la longitud del próximo *período de ejecución* y distribuir los créditos asignados a una aplicación entre sus hilos.

En el primer paso, se calcula el peso dinámico de cada aplicación en el sistema, utilizando

Algoritmo 4.1: Algoritmo Prop-SP

{ • *R conjunto de aplicaciones activas.*
• N_{FC} *número de cores rápidos (FCs).*
• CRED_1FC_REF *cantidad de créditos consumidos en cualquier core rápido durante un período de ejecución usado como referencia.*
• cred_per_fc_next_period *cantidad de créditos consumidos en cualquier core rápido durante el próximo período de ejecución.* }

S:=[]; total_weight:=0; total_credits:=CRED_1FC_REF $* N_{FC}$;

{ **PASO 1 ⇒ Calcula el peso dinámico de cada aplicación y la suma de los pesos dinámicos de todas las aplicaciones.** }

para cada app *in* R **hacer**
    speedup_{app}:= estima el speedup de app;
    $\text{dyn\_weight}_{app} := (\text{speedup}_{app} - 1) * \text{static\_weight}_{app}$;
    total_weight := total_weight + dyn_weight_{app} ;
    Insertar app en S tal que S esté ordenado en orden descendiente por dyn_weight_{app} ;
fin

{ **PASO 2 ⇒ Asignar créditos a cada aplicación en base a** dyn_weight_{app} }

para cada app *en* S **hacer**
    $$\text{credit}_{app} := \frac{\text{total\_credits} * \text{dyn\_weight}_{app}}{\text{total\_weight}};$$
fin

{ **PASO 3 ⇒ Determinar la longitud del próximo período de ejecución y distribuir los créditos asignados a una aplicación entre sus hilos.** }

Calcular cred_per_fc_next_period ;
scale_factor:=cred_per_fc_next_period/CRED_1FC_REF;
para cada app *en* S **hacer**
    credit_{app}:=credit_{app} * scale_factor;
    Distribuir credit_{app} entre los hilos de app
fin

la siguiente ecuación:

$$Dyn\_weight_{app} = (Speedup - 1) \cdot Static\_weight_{app} \tag{4.1}$$

Luego, se suman los pesos dinámicos de todas las aplicaciones, y se almacenan en la variable *Total_weight*:

$$Total\_weight = \sum_{app}^{apps} Dyn\_weight_{app} \tag{4.2}$$

En el segundo paso, se asignan créditos a cada aplicación en función de su peso dinámico. Los créditos asignados (a una aplicación) se calculan como sigue:

$$Credits_{app} = \frac{Total\_credits \cdot Dyn\_weight_{app}}{Total\_weight} \tag{4.3}$$

Donde $Total\_credits$ se calcula como:

$$Total\_credits = CRED\_FC\_REF \cdot N_{FC} \tag{4.4}$$

Donde N_{FC} es la cantidad de cores rápidos en el sistema. *CRED_FC_REF* es un valor que indica la cantidad de créditos consumidos en cada core rápido durante un *período de ejecución* de longitud constante usado como referencia. La distribución de créditos realizada en este paso se lleva a cabo asumiendo un *período de ejecución* de referencia fijo. Esto se debe a que la longitud del próximo *período de ejecución* (*cred_per_fc_next_period*) se calcula en el tercer paso para controlar la tasa de migración, tema que elaboraremos en la sección 4.1.2.

En el tercer y último paso, además de determinar la longitud del próximo *período de ejecución*, los créditos asignados a la aplicación se escalan usando el siguiente factor:

$$Scale\_factor = \frac{cred\_per\_fc\_next\_period}{CRED\_FC\_REF} \tag{4.5}$$

A continuación el número de créditos asignados a una aplicación app se calcula como:

$$Credits_{app} = Credits_{app} \cdot Scale\_factor \tag{4.6}$$

Por último, los créditos otorgados a una aplicación se distribuyen entre sus hilos activos. Para aplicaciones secuenciales, el proceso de asignación de créditos asigna todos los créditos al único hilo existente. Para aplicaciones multi-hilo, Prop-SP implementa tres estrategias de

distribución creditos que explicaremos en la sección 4.2.

4.1.2. Intercambio de hilos entre cores

El mecanismo de intercambio de hilos entre cores asegura que aquellos hilos que tienen asignados créditos de cores rápidos tengan la oportunidad de consumirlos, sin afectar al balance de carga del sistema.

Para ilustrar este mecanismo, supongamos un AMP que integra un core rápido y un core lento. Consideremos además, que hay dos hilos ejecutando en el sistema, T_A y T_B, ambos poseen créditos de cores rápidos, cada uno asignado a cores diferentes para preservar el balance de carga (T_A asignado al core rápido y T_B asignado al core lento).En algún momento el hilo que se ejecuta en el core rápido (T_A) agotará todos sus créditos. En ese instante Prop-SP activará el mecanismo que intercambia los hilos entre los cores. A partir de ese momento, el hilo T_A se ejecutará en el core lento, mientras que T_B se ejecutará en el core rápido. De esta manera, el hilo T_B tendrá la oportunidad de consumir sus créditos en el core rápido mientras se mantiene el balance de carga en el sistema.

Para asegurar que el intercambio de hilos [1] se realiza a una frecuencia media controlada durante el *período de ejecución*, la longitud de este período debe ajustarse durante el proceso de asignación de créditos que lo precede. Es necesario hallar un equilibrio entre el *período de ejecución* y la frecuencia de intercambio de hilos. Básicamente, cuanto más largo es el período de ejecución, la frecuencia de intercambio de hilos es más baja (overheads de migración más bajos). Por el contrario, períodos de ejecución más cortos hacen que el planificador Prop-SP sea más consciente de las fases de programa, siendo capaz de reaccionar a fases de *speedup* de grano fino. Sin embargo, esto puede incrementar el *overhead* relacionado a las migraciones.

Para encontrar un equilibrio entre la longitud del *período de ejecución* y la tasa de migraciones, realizamos un estudio experimental en las plataformas utilizadas para la evaluación. Al realizar dicho estudio encontramos que establecer la longitud del *período de ejecución* tal que las migraciones se realizan en promedio cada 850ms, permite reducir el *overhead* de migración sustancialmente y al mismo tiempo hace posible capturar la mayoría de las fases de *speedup* de grano grueso.

Los mecanismos descritos hasta el momento no garantizan que los hilos asignados a cores del mismo tipo reciban una cantidad justa de tiempo de CPU y tampoco aseguran el balance de carga en el AMP. Para hacer esto posible, Prop-SP emplea políticas de balance de carga convencionales implementadas en los sistemas operativos actuales sobre hardware simétrico, estas políticas garantizan el balance de carga y mantienen la justicia en cores del mismo tipo. Para los experimentos que se muestran en este capítulo, Prop-SP se implementó

[1] El número de intercambios de hilos para lograr una distribución justa de los ciclos de core rápido en el período de ejecución puede determinarse inmediatamente después de repartir créditos. Básicamente, un hilo no será intercambiado si alguna de las siguientes condiciones es verdadera: (1) el hilo no recibió créditos o (2) el hilo recibió créditos suficientes para ejecutar en un core rápido durante todo el período de ejecución.

sobre el planificador de OpenSolaris en el framework de planificación usado en esta tesis. Recordemos que todos los cores del sistema se organizan en dos particiones de cores: una partición para los cores rápidos y otra para los cores lentos. OpenSolaris asegura la justicia y el balance de carga entre los cores de la misma partición, mientras que Prop-SP se encarga del balance de carga entre particiones. Para lograr este objetivo, Prop-SP mantiene la carga de las distintas *run queues* y asegura que los cores lentos nunca reciben más carga que los cores rápidos. Asimismo, Prop-SP migra hilos desde cores rápidos a cores lentos cuando estos últimos quedan ociosos, maximizando la utilización de los cores rápidos.

4.2. Soporte para aplicaciones multi-hilo en Prop-SP

El planificador Prop-SP realiza la distribución de créditos de core rápido entre aplicaciones. Luego, los créditos otorgados a una aplicación deben distribuirse entre sus hilos activos utilizando alguna estrategia de distribución. Intuitivamente, si incrementamos el número de créditos de cores rápidos asignados a una aplicación multi-hilo (por ejemplo, como resultado de aumentar su prioridad en una carga de trabajo multiprogramada), el rendimiento de la aplicación también debe incrementarse. Esto se debe a que la aplicación recibirá una porción mayor de tiempo de core rápido. El objetivo principal del sistema de distribución de créditos de Prop-SP es lograr que esto suceda.

Para llevar a cabo este objetivo Prop-SP soporta tres estrategias de distribución de créditos:

- **Even**: distribuye los créditos de manera uniforme entre todos los hilos de la aplicación.
- **BusyFCs**: va asignando a cada hilo de la aplicación la cantidad máxima de créditos que estos pueden consumir en el próximo *período de ejecución* (*cred_per_fc_next_period*). Esta asignación se realiza hasta que no haya más créditos por distribuir. Por lo tanto, es posible que algunos hilos no reciban créditos. Esta estrategia tiende a favorecer hilos que recibieron créditos en asignaciones previas.
- **AID** (*Asymmetric Iteration Distribution*): es la combinación de una estrategia de distribución de créditos y un mecanismo de interacción *runtime system-SO* adaptado a aplicaciones OpenMP regulares.

Las aplicaciones OpenMP están compuestas por varios bucles *do-all* ejecutados uno después de otro y separados por barreras de sincronización implícitas. Generalmente, las iteraciones de los bucles se distribuyen entre los hilos disponibles por igual para lograr un balance de carga adecuado. Con esta distribución de iteraciones, es posible que los hilos lleguen aproximadamente al mismo tiempo a la barrera de sincronización. Sin embargo, en un AMP la asignación de la misma cantidad de trabajo entre hilos puede desencadenar desbalance de carga, ya que el planificador del sistema operativo puede asignar hilos de la aplicación a diferentes tipos de core. Los hilos que se ejecutan en cores rápidos suelen

completar su parte de las iteraciones del bucle más rápido que los hilos asignados a cores lentos. Por esta razón, los hilos asignados a cores rápidos quedarán ociosos, en forma activa o bloqueante, esperando en la barrera de sincronización hasta que los hilos en cores lentos completen su parte. Una forma posible de abordar este problema es recurrir a los mecanismos de distribución de iteraciones dinámicos provistos por OpenMP. Desafortunadamente, el *overhead* en la asignación de iteraciones dinámica puede ser significativo y el comportamiento impredecible de este enfoque tiende a degradar la localidad de los datos [7].

Para superar esta limitación, proponemos una estrategia alternativa de distribución de iteraciones de bucles para sistemas AMP que llamamos AID (*Asymmetric Iteration Distribution*). Esta estrategia se basa en la interacción entre el *runtime system* (en este caso de OpenMP) y el planificador del sistema operativo. La idea general de AID es la siguiente: al comienzo de cada bucle paralelo, el *runtime system*, que se ejecuta en el espacio de usuario, tiene como objetivo distribuir las iteraciones del bucle de manera que los hilos lleguen a la barrera de sincronización al mismo tiempo. Para realizar esta distribución, el planificador comunica al *runtime system* el número de hilos que una aplicación tiene asignado a cores rápidos en ese momento ($N_{FC\_threads}$) y una estimación de cuan rápido los hilos de la aplicación retiran instrucciones sobre cores rápidos con respecto a los cores lentos. Esto último es el *speedup factor* de los hilos de la aplicación.

Para evitar el *overhead* asociado a la interacción entre el *runtime system* y el sistema operativo, el planificador mantiene actualizados los valores $N_{FC\_threads}$ y SF en una región de memoria compartida entre el espacio de usuario y el espacio del kernel. Para ilustrar cómo el *runtime system* realiza la distribución de iteraciones introducimos la siguiente notación:

- SPI_{FC}, SPI_{SC}: Promedio de segundos por instrucción alcanzados por los hilos de una aplicación cuando se ejecutan en un core rápido y un core lento, respectivamente. El SF en función de estos valores es $SF = \frac{SPI_{sc}}{SPI_{fc}}$.
- NI: Número total de instrucciones (dinámicas) en el bucle paralelo.
- f_{FC}, f_{SC}: Fracción de las instrucciones de NI que los hilos asignados a cores rápidos y cores lentos tienen que completar para alcanzar la barrera de sincronización, respectivamente.

Suponemos que $T_{barrier}$ es la cantidad de tiempo que se demoran los hilos en completar su parte del bucle paralelo. Si los hilos que se ejecutan en cores rápidos y cores lentos llegan a la barrera al mismo tiempo, entonces concluimos lo siguiente:

$$T_{barrier} = NI \cdot f_{FC} \cdot SPI_{FC} = NI \cdot f_{SC} \cdot SPI_{SC} \;\Rightarrow\; f_{FC} = \frac{NI \cdot f_{SC} \cdot SPI_{SC}}{NI \cdot SPI_{FC}} = SF \cdot f_{SC} \tag{4.7}$$

Debido a que la distribución de iteraciones de los bucles paralelos no se puede realizar a nivel de instrucción, aproximamos la asignación ideal propuesta en la ecuación 4.7 de la siguiente manera: Si k es el número de iteraciones asignado a los hilos sobre cores lentos, entonces a los hilos que se ejecuten en cores rápidos les serán asignados $SF \cdot k$ iteraciones.

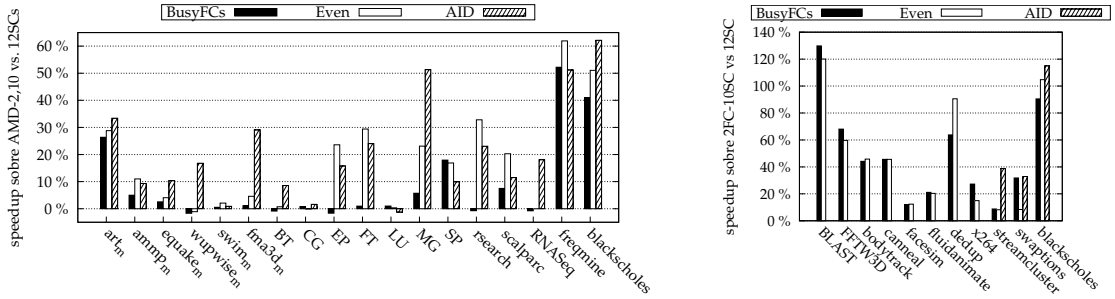
Aunque el *runtime system* conoce el número de hilos asignados a los cores rápidos en un momento dado, no sabe exactamente cuales son esos hilos. Para asegurar que los hilos que reciben más iteraciones coinciden con aquellos asignados a cores rápidos por el planificador, por convención, el *runtime system* asigna más iteraciones a los hilos con $TID(IDs\ de\ hilos)\ \epsilon\ (0..N_{FC\_threads}-1)$. Al mismo tiempo, Prop-SP distribuye créditos de core rápido utilizando un enfoque BusyFCs que favorece hilos con más bajo TID.

Para implementar AID, modificamos el *runtime system* de OpenMP incluido en el compilador GCC versión 4.5. El compilador GCC también se modificó para asegurar que el código multihilo generado a partir de OpenMP invoque una función específica del *runtime system* al comienzo del bucle paralelo, y así llevar a cabo la distribución de iteraciones en un AMP. Aunque para usar AID es necesario compilar una aplicación OpenMP, no es preciso llevar a cabo ninguna modificación en el código fuente de la misma.

Para nuestros experimentos configuramos el *runtime system* de OpenMP para que utilizara sincronización adaptativa, lo que resulta beneficioso para AID cuando la aplicación se ejecuta sola en el sistema. En el caso de que el sistema operativo obtenga una predicción del SF inferior que el valor real, los hilos asignados a cores rápidos alcanzarán la barrera de sincronización implícita al final del bucle paralelo antes que los hilos en cores lentos. Empleando la sincronización adaptativa, estos hilos hacen espera activa por un tiempo y luego se bloquean dejando los cores rápidos ociosos. Como Prop-SP intenta mantener los cores rápidos ocupados, elegirá un hilo que se encuentre ejecutando en un core lento y lo migrará al core rápido. De esta forma es posible mejorar el rendimiento de la aplicación.

Aunque AID está pensado para aplicaciones OpenMP de paralelismo de datos, otros tipo de aplicaciones también pueden beneficiarse, por ejemplo programas POSIX-threads. Sin embargo, para esto es preciso modificar el código fuente. A pesar de esto, los resultados de la sección 4.2.1 muestran que AID resulta efectivo tanto para aplicaciones OpenMP como para aplicaciones POSIX.

Además de estar equipado con diferentes estrategias de distribución de créditos, Prop-SP incluye soporte adicional para acelerar fases secuenciales de aplicaciones paralelas. Trabajos previos demostraron que las aplicaciones paralelas (durante las fases secuenciales y regiones críticas de grano grueso), generalmente exponen al sistema operativo un sólo hilo activo [3], [54]. Para acelerar fases de paralelismo a nivel de hilo (TLP) ejecutándolas en cores rápidos, Prop-SP asegura que los créditos de core rápido no consumidos por los hilos bloqueados pueden ser consumidos por hilos activos de la misma aplicación. En particular, durante las fases secuenciales todos los créditos de core rápido pertenecientes a la aplicación se ponen a disposición del único hilo en ejecución.



(a) Aplicaciones OpenMP

(b) Aplicaciones POSIX-threads

Figura 4.1: Speedup alcanzado por distintas aplicaciones paralelas ejecutadas en una configuración asimétrica 2F-10S con respecto a usar sólo cores lentos.

4.2.1. Efectividad de las distintas estrategias de distribución de créditos

En esta sección evaluamos experimentalmente las distintas estrategias de distribución de créditos. Para la evaluación utilizamos algunas aplicaciones HPC de las suites de benchmarks SPEC CPU, NAS, PARSEC y Minebench.

La figura 4.1 muestra el *speedup* que experimentan estas aplicaciones, al ejecutarse con doce hilos en un sistema AMP compuesto por 2 cores rápidos y 10 cores lentos (2F-10S). Para este análisis, emulamos un AMP sobre la plataforma simétrica AMD-12 descrita en la sección 2.1. El *speedup* que se muestra en la figura 4.1 es aquel obtenido con respecto al rendimiento observado cuando la aplicación se ejecuta en un sistema con 12 cores lentos.

Las aplicaciones OpenMP de la figura 4.1a se evaluaron empleando todas las estrategias de distribución de créditos, mientras que la mayoría de las aplicaciones POSIX-threads de la figura 4.1b se evaluaron empleando sólo Even y BusyFCs. Como se mencionó anteriormente, para aplicar AID en aplicaciones POSIX es necesario modificar el código fuente. Sin embargo, a modo de prueba modificamos tres aplicaciones PARSEC (*streamcluster*, *swaptions* y *blackholes*) para que soporten AID.

Cuando una aplicación paralela se ejecuta sola en un sistema AMP bajo la distribución de créditos Even, Prop-SP distribuye el tiempo de cores rápidos entre todos los hilos realizando intercambio de hilos periódicos. En nuestra configuración este intercambio se realiza aproximadamente cada 850ms. Los resultados revelan que Even resulta muy beneficioso para aplicaciones de paralelismo de grano medio o grueso, tales como *EP*, *FT*, *Rsearch*, *scalparc* o *freqmine*. Por el contrario, en aplicaciones OpenMP con paralelismo de grano fino como *wupwise*, *fma3d* o *RNASeq*, muchos bucles paralelos pueden ejecutarse cada segundo. En este caso, los hilos que sufren intercambios de tipo de core en intervalos de grano grueso, hacen que aquellos hilos que se ejecutan en cores rápidos esperen a los demás en las barreras de sincronización, utilizando estos cores de forma ineficiente. En este escenario, observamos que incrementar la tasa de intercambio de hilos era contraproducente en la mayoría de los

casos debido al aumento del *overhead* asociado a las migraciones. Para estas aplicaciones, AID es la estrategia de distribución de créditos que otorga el mejor rendimiento. Nótese que AID no desencadena intercambios periódicos de hilos; en este escenario, se asignan N_{FC} hilos a cores rápidos.

En algunos casos, AID obtiene un rendimiento menor de lo esperado debido a fallos de predicción de *SF*. Éste es el caso de aplicaciones como *EP*, *Rsearch* o *scalparc* que incluyen un sólo bucle paralelo. Estos programas presentan un perfil de rendimiento muy diferente en el código que precede al bucle que en el código del cuerpo del bucle. Esto lleva a imprecisiones en la estimación de *SF* y, a su vez, genera un desbalance en la distribución de la carga de trabajo entre los hilos de la aplicación. Este problema podría mitigarse dedicando unas pocas iteraciones para *profiling* y así obtener una estimación más precisa del *SF*. A continuación, se realizaría la distribución de la carga de trabajo para las restantes iteraciones en base al valor del *SF* obtenido.

En [54], los autores muestran que ejecutar una única aplicación en un AMP bajo un planificador que mantiene los cores rápidos lo más ocupado posible, resulta beneficioso para aplicaciones paralelas con partes secuenciales significativas. En esta categoría se incluyen aplicaciones como *art_m*, *freqmine*, *blackscholes*, *blast*, *FFTW3D*, o *bodytrack*, que experimentan valores altos de *speedup* bajo todos los algoritmos de distribución de créditos disponibles (incluyendo BusyFCs). Intuitivamente, estas aplicaciones atraviesan fases de ejecución que presentan paralelismo a nivel de hilo reducido, con sólo unos pocos hilos en ejecución. Como Prop-SP intenta mantener los cores rápidos ocupados, los pocos hilos activos se asignan a los cores rápidos ociosos acelerando estas fases de programa. En general, al usar BusyFCs, el algoritmo Prop-SP sólo migra hilos a cores rápidos cuando éstos pasan a estar ociosos, por lo que esta estrategia de distribución de créditos está sujeta a un *overhead* de migración menor que Even. Esto permite a BusyFCs superar a Even incluso para aplicaciones con fases secuenciales considerables como *SP*, *blast* o *FFTW3D*.

Como los resultados revelan, los programas POSIX que utilizan AID (los tres últimos en la figura 4.1b) obtienen el mayor rendimiento. Esto se debe al uso de una distribución de carga de trabajo y una estrategia de balance de carga asimétrica coordinada con el planificador que favorece a los hilos con los TIDs más bajos.

Claramente, las tres estrategias de distribución de créditos cumplen con las necesidades de diferentes tipos de aplicaciones. Por otra parte, estas estrategias podrían también resultar adecuadas para otros modelos de aplicación paralela no exploradas en esta tesis.

4.2.2. Determinando el speedup de las aplicaciones en tiempo de ejecución

Para asignar créditos a una aplicación, Prop-SP tiene en cuenta el beneficio relativo (*speedup*) que una aplicación obtiene al ejecutar todos sus hilos en cores rápidos con respecto a hacerlo en los cores lentos. Para aplicaciones secuenciales este beneficio coincide con el

Tabla 4.1: Fórmulas para estimar el speedup para una aplicación multi-hilo bajo las distintas estrategias de distribución de crédito. N es el número de hilos en la aplicación, N_{FC} es el número de cores rápidos en el AMP y SF es el speedup factor promedio de los hilos de la aplicación.

Distribución de créditos	Ecuación
Even	$SP_{Even} = \frac{MIN(N_{FC}, N)}{N} \cdot (SF-1) + 1$ (4.8)
BusyFCs	$SP_{BusyFCs} = \frac{SF-1}{(\frac{N-1}{N_{FC}}+1)^2} + 1$ (4.9)
AID	$SP_{AID} = \frac{MIN(N_{FC}, N)}{N} \cdot (SF-1) + 1$ (4.10)

speedup factor del único hilo en ejecución. Sin embargo, el *speedup factor* no aproxima este beneficio para aplicaciones multi-hilo. Para obtener una estimación del *speedup* para una aplicación multi-hilo, es necesario considerar otros factores además del *speedup factor* de los hilos individuales, como: su grado de paralelismo a nivel de hilo, el número de cores rápidos de la arquitectura o cómo los créditos de cores rápidos se distribuyen entre sus hilos.

La tabla 4.1 muestra las ecuaciones utilizadas por Prop-SP para estimar el *speedup* de las aplicaciones multi-hilo bajo las distintas estrategias de distribución de créditos. El anexo A ilustra todo el proceso de derivación de estas fórmulas. Debido a las suposiciones utilizadas en nuestro análisis, que se indican en el anexo, el proceso de derivación asociado a las estrategias Even y AID nos llevan a la misma fórmula.

Para determinar el *speedup factor* de un hilo en tiempo de ejecución, Prop-SP utiliza un modelo de estimación específico de la plataforma. Este modelo recolecta los valores de distintas métricas de rendimiento monitorizadas durante la ejecución de las aplicaciones (como por ejemplo: IPC o la tasa de fallos de último nivel de caché). La técnica para la construcción de modelos de estimación de *SF* utilizada en este capítulo se propuso en [53]. Esta técnica genera modelos de estimación precisos y sencillos para entornos AMP que difieren sólo en la frecuencia del procesador o sólo exhiben ligeras diferencias en la microarquitectura, como los usados en este capítulo. En nuestra configuración, las muestras de los contadores de rendimiento se capturan por hilo cada 200ms. Empleando este intervalo de muestreo observamos que el *overhead* asociado al muestreo y a la evaluación del modelo de estimación es insignificante.

4.3. Otros algoritmos analizados en este capítulo

Para evaluar la efectividad de Prop-SP, comparamos este algoritmo con otros cuatro estrategias de planificación para AMP: RR [8], A-DWRR [33] , HSP [28, 54] y CAMP [53]. Estas estrategias de planificación (a excepción de CAMP) se describieron en el capítulo 3. Para la evaluación experimental de este capítulo, implementamos todos los algoritmos en el kernel de OpenSolaris.

CAMP [53] intenta optimizar el rendimiento de las aplicaciones asignando a cores rápidos aquellas con un *speedup* alto, pero sin tener en cuenta las prioridades. A diferencia de HSP, CAMP clasifica los hilos de las aplicaciones en varias clases o *bins* (*high*, *medium*, *low* y *very low*) de acuerdo al *speedup* estimado de la aplicación, al cual nos referimos como el *factor de utilidad*. Este factor depende del número de hilos de una aplicación que se encuentran en ejecución y de la estimación del *SF* de estos hilos en tiempo de ejecución. Los límites entre las clases de hilos se establecen mediante tres umbrales específicos de plataforma. Los hilos que pertenecen a la clase *high* se asignan a cores rápidos; si aún existen cores rápidos disponibles se asignan hilos de clase *medium*, *low* y *very low*, en ese orden. Si el número de hilos de la clase *high* es mayor al número de cores rápidos, CAMP distribuye el tiempo de core rápido entre estos hilos utilizando un mecanismo *round robin*. Este mecanismo ayuda a reducir la falta de justicia en comparación con la ejecución sobre cores rápidos de sólo unos pocos hilos de clase *high*.

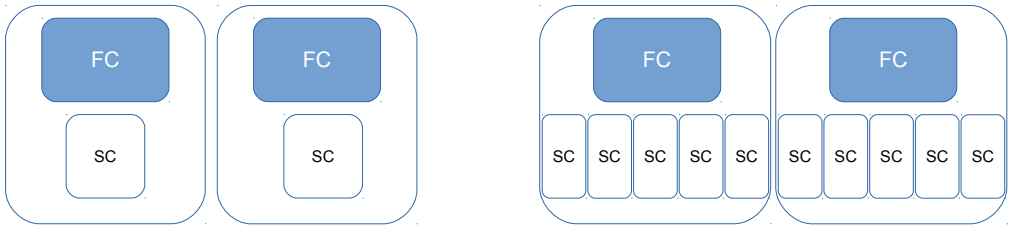
De ahora en adelante, llamaremos *Prop-SP (dynamic)* a la versión de Prop-SP que estima el *speedup factor* de las aplicaciones en tiempo de ejecución utilizando modelos de estimación. Debido a que los modelos de estimación no garantizan predicciones perfectas, optamos por comparar este algoritmo contra una versión estática de Prop-SP que llamaremos *Prop-SP (static)*. En la versión estática, el *speedup factor* de cada aplicación se obtiene *offline* y representa el ratio entre los tiempos de ejecución de la aplicación en el core rápido y en el core lento. Para aplicaciones multi-hilo, aproximamos el *speedup factor* ejecutándolas con un sólo hilo en cada tipo de core. Nótese que Prop-SP (static) calcula el *speedup* de las aplicaciones mediante el uso de las ecuaciones en la tabla 4.1, pero asumiendo un *speedup factor* estático.

La implementación base de RR no ofrece soporte a prioridades de usuario. Para llevar a cabo nuestra evaluación experimental agregamos a este algoritmo el soporte de prioridades, garantizando que cada aplicación recibe una porción de tiempo de core rápido proporcional a su prioridad. Esta fracción de tiempo se distribuye equitativamente entre todos los hilos de la aplicación.

4.4. Evaluación experimental

Nuestro análisis experimental se llevó a cabo emulando un AMP sobre hardware multicore simétrico mediante la reducción de la frecuencia de un subconjunto de cores. Específicamente, se utilizaron las plataformas Intel-8 y AMD-12 descritas en la sección 2.1. Asimismo, utilizamos cargas de trabajo multi-aplicación formadas por aplicaciones de diversas suites de benchmarks para HPC: SPEC CPU2006 y CPU2000, OMP PARSEC, NAS Benchmarks y Minebench. También experimentamos con *BLAST* (un benchmark para bioinformática) y *FFTW3D* (un benchmark HPC que realiza la transformada rápida de Fourier).

Para evaluar los diferentes algoritmos de planificación, utilizamos tres configuraciones AMP cuya topología se muestra en la figura 4.2.



(a) Intel-2,2 y AMD-2,2 (b) AMD-2,10

Figura 4.2: Configuración AMP.

1. **Intel-2,2** (figura 4.2a): dos cores rápidos y dos cores lentos sobre Intel-8 organizados en dos *cluster* con un core rápido y un core lento cada uno.

2. **AMD-2,2** (figura 4.2a): compuesto por dos *cluster* con un core rápido y un core lento cada uno sobre AMD-12.

3. **AMD-2,10** (figura 4.2b): dos *cluster* con un core rápido y cinco cores lentos cada uno sobre AMD-12.

Las topologías han sido cuidadosamente elegidas para reducir la contención de los recursos compartidos tanto como sea posible. Además, en estas configuraciones el planificador del sistema operativo puede realizar migraciones de hilos menos costosas entre distintos tipos de cores que comparten el último nivel de caché (LLC).

Todas las cargas de trabajo que exploramos tienen tantos hilos como cores hay en la plataforma. Asimismo, al ejecutar cargas de trabajo multiprogramadas garantizamos que todas las aplicaciones se inician simultáneamente y, cuando una aplicación termina, se vuelve a ejecutar tantas veces como sea necesario hasta que la aplicación de mayor duración del conjunto se completa en tres ocasiones. Para cada carga de trabajo bajo un algoritmo de planificación determinado, medimos el tiempo de ejecución de cada una de las aplicaciones. Luego, para cada aplicación calculamos la media geométrica de sus tiempos de ejecución; y por último, determinamos el ASP y *unfairness* para el algoritmo de planificación en cuestión.

El resto de esta sección se divide en dos partes. En la sección 4.4.1 analizamos los distintos algoritmos de planificación, para cargas de trabajo multi-aplicación compuestas por aplicaciones con la misma prioridad. La sección 4.4.2 cubre escenarios multi-aplicación en los que se asignan diferentes prioridades a las aplicaciones.

4.4.1. Aplicaciones con la misma prioridad

En esta sección evaluamos dos conjuntos de cargas de trabajo. Las primeras formadas por una combinación de aplicaciones secuenciales y aplicaciones multi-hilo. Las segundas

Tabla 4.2: Cargas de trabajo multi-aplicación compuestas por programas secuenciales y multi-hilo

Categorias	**Benchmarks**
3STH-1HPH	`hmmer`, `gobmk`, `h264ref`, `fma3d_m`(9)
3STH-1HPL	`povray`, `gamess`, `gobmk`, `swim_m`(9)
2STH-1PSH-1HPM	`gamess`, `bzip2`, `blast`(4), `wupwise_m`(6)
1STH-1STM-1STL-1PSH	`gamess`, `astar`, `soplex`, `blackscholes`(9)
1PSH-1PSL	`semphy`(6), `FFTW3D`(6)
2PSH-1HPM	`blast`(4), `semphy`(4), `wupwise_m`(4)
1PSH-1HPL	`semphy`(6), `equake_m`(6)
1HPH-1HPL	`fma3d_m`(6), `equake_m`(6)
1PSH-1HPH	`blackscholes`(6), `fma3d_m`(6)

compuestas únicamente por aplicaciones secuenciales.

4.4.1.1. Escenario de aplicaciones secuenciales y multi-hilo

En este escenario, evaluamos nueve cargas de trabajo que combinan aplicaciones secuenciales (de un único hilo) y multi-hilo que cubren un amplio espectro de *speedups*. Para la creación de las cargas de trabajo, clasificamos las aplicaciones de acuerdo a su grado de paralelismo en tres categorías:

- *High Parallel* o HP: altamente paralelas
- Parcialmente secuenciales o PS: aplicaciones paralelas con un componente secuencial mayor al 25 % del tiempo total de ejecución
- *Single threaded* o ST: secuenciales

A su vez, dividimos las categorías anteriores de acuerdo a su *SF* en tres subgrupos:

- H: *SF* alto.
- M: *SF* medio.
- L: *SF* bajo.

Las combinaciones de las aplicaciones seleccionadas, que se muestran en la tabla 4.2, representan escenarios con diferentes grados de competencia por los escasos cores rápidos del AMP. Estas cargas de trabajo incluyen aplicaciones de todas las clases posibles (paralelismo/*SF*); excepto de una aplicación parcialmente secuencial con *SF* medio (PSM) que no encontramos en ninguna de las suites de benchmarks exploradas.

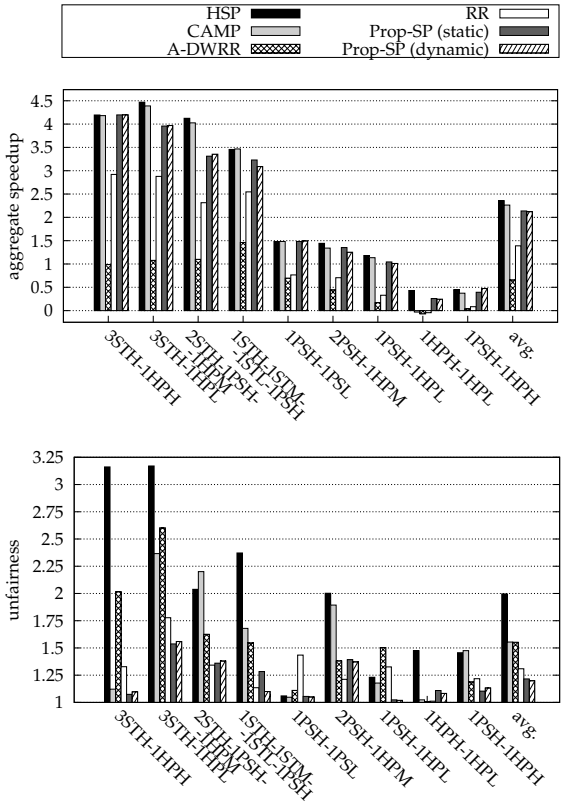


Figura 4.3: Resultados sobre la plataforma AMD-2,10

Las categorías de aplicaciones se muestran en la tabla 4.2 en el mismo orden en el que figuran los benchmarks correspondientes. Por ejemplo: en la categoría 1PSH-1HPL, *semphy* es la aplicación PSH y *equake_m* es la aplicación HPL. En aplicaciones multi-hilo, el número entre paréntesis que sigue al nombre de cada aplicación indica el número de hilos con los que se ejecuta. Para obtener CT_{fast} para una aplicación en particular (necesario para calcular *unfairness* - ecuación 3.3), consideramos el tiempo de ejecución cuando ésta se ejecuta sola en el AMP con la mejor estrategia de distribución de créditos por hilo para esta aplicación (según lo visto en la sección 4.2). Esta estrategia de distribución de créditos también se utiliza al ejecutar la aplicación bajo Prop-SP y HSP.

La figura 4.3 muestra los resultados obtenidos para las cargas de trabajo ejecutadas sobre AMD-2,10.

En los resultados se puede observar un amplio rango en los valores de ASP obtenidos. Esto se debe a la combinación de aplicaciones paralelas, que no obtienen ninguna aceleración al usar cores rápidos (como aplicaciones HPL), con otras aplicaciones que experimentan mejoras de rendimiento significativas a partir del uso de este tipo de core (como las aplicaciones secuenciales). Se puede observar además, que las cargas de trabajo que no incluyen ninguna aplicación secuencial exhiben un valor de ASP muy bajo en todos los algoritmos de planificación.

En la mayoría de los casos el algoritmo HSP consigue el valor más alto de ASP. Sin embargo, obtiene los peores valores de *unfairness* (mientras más alto peor) para la mayoría de las cargas de trabajo. CAMP, que al igual que HSP intenta maximizar el rendimiento, obtiene valores de ASP similares que este algoritmo para la mayoría de las aplicaciones. Sin embargo, el hecho que CAMP reparte el tiempo de cores rápidos entre los hilos que se encuentran en ejecución, contribuye a reducir la injusticia en escenarios donde el número de aplicaciones con *speedup* alto es mayor que el número de cores rápidos, como 3STH-1HPH o 3STH-1HPL. A pesar de este hecho, CAMP obtiene valores de *unfairness* considerablemente más altos que Prop-SP en la mayoría de los casos.

En cuanto a RR y A-DWRR , podemos ver que ninguno de estos algoritmos ofrece un rendimiento comparable a los otros algoritmos de planificación. En general, ambos hacen un uso ineficiente de los cores rápidos de la plataforma, especialmente en las últimas cinco cargas de trabajo. Esto de debe principalmente a que no tienen en cuenta el *speedup* de las aplicaciones.

El algoritmo A-DWRR se esfuerza por garantizar que cada hilo reciba el mismo tiempo de CPU escalado, independientemente de la aplicación a la cual pertenece. Por esta razón, las aplicaciones con un gran número de hilos reciben una fracción de tiempo de core rápido mayor que otras, haciendo que estas últimas puedan degradar su rendimiento significativamente.

RR, por otro lado, comparte de manera equitativa los ciclos de core rápido por aplicación, lo que conduce a una utilización más eficiente del AMP y por lo tanto reduce el*unfairness* en la mayoría de los casos.

En este escenario, nuestro algoritmo de planificación Prop-SP es capaz de igualar el rendimiento de HSP y CAMP para 3STH-1HPH, 1PSH-1PSL y 1PSH-1HPH, mientras alcanza valores cercanos para el resto de las cargas de trabajo. Al mismo tiempo, obtiene valores de *unfairness* muy inferiores a HSP, CAMP y A-DWRR en la mayoría de los casos, pero además muestra valores comparables, y en muchos casos más bajos, que RR. En particular, *Prop-SP (static)* y *Prop-SP (dynamic)* exhiben un comportamiento muy similar, a pesar de las imprecisiones en el modelo de estimación de *SF* diseñado para la plataforma AMD.

4.4.1.2. Escenario de aplicaciones secuenciales

En este escenario construimos 10 cargas de trabajo combinando 18 aplicaciones secuenciales de la suite de benchmarks SPEC CPU2006. Clasificamos las aplicaciones en tres categorías en función de su *SF*:

- H: *SF* alto.
- M: *SF* medio.
- L: *SF* bajo.

Tabla 4.3: Cargas de trabajo multi-aplicación compuestas por aplicaciones secuenciales

Categorías	Benchmarks
4H	`povray`, `gobmk`, `bzip2`, `sjeng`
3H-1M	`povray`, `h264ref`, `perlbench`, `astar`
3H-1L_A	`hmmer`, `namd`, `perlbench`, `soplex`
3H-1L_B	`hmmer`, `h264ref`, `gobmk`, `milc`
2H-2M_A	`povray`, `bzip2`, `leslie3d`, `sphinx3`
2H-2M_B	`gamess`, `gobmk`, `xalancbmk`, `astar`
2H-2L_A	`hmmer`, `gobmk`, `lbm`, `soplex`
2H-2L_B	`povray`, `h264ref`, `lbm`, `omnetpp`
1H-1M-2L	`sjeng`, `leslie3d`, `lbm`, `soplex`

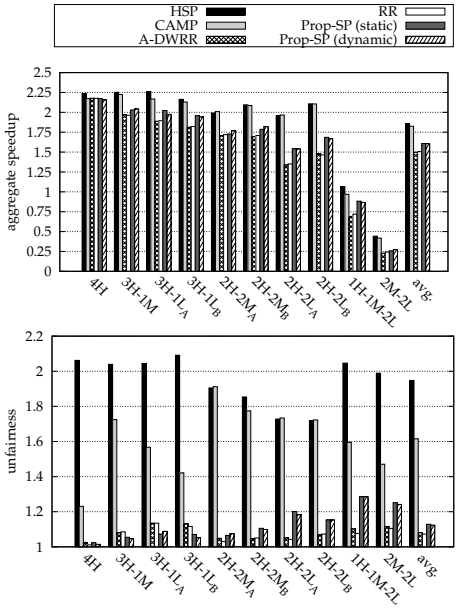


Figura 4.4: Resultados sobre la plataforma Intel-2,2

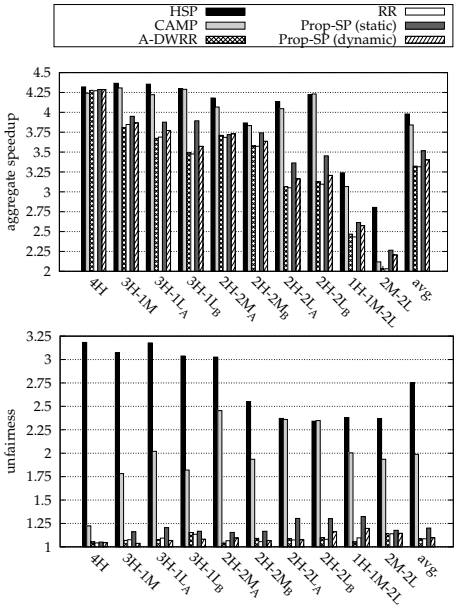


Figura 4.5: Resultados sobre la plataforma AMD-2,2

La tabla 4.3 muestra las diez cargas de trabajo seleccionadas. Las cargas de trabajo aparecen ordenadas en orden descendente por el número de aplicaciones con alto *SF* que las componen, y luego por el número de aplicaciones con *SF* medio y/o *SF* bajo. Por ejemplo: la carga de trabajo 4H está compuesta por 4 aplicaciones secuenciales con *SF* alto. A continuación, en la categoría 3H-1M, *povray*, *h264ref* y *perlbench* son las tres aplicaciones con *SF* alto (3H) y *astar* es la aplicación con *SF* medio (1M).

Para distinguir entre cargas de trabajo compuestas por exactamente las mismas categorías de referencia, se añadió un sufijo al nombre. Por ejemplo: existen dos cargas de trabajo en la categoría 3H-1L, la primera lleva el subfijo _A (3H-1L_A) y la segunda el subfijo _B (3H-1L_B).

Las figuras 4.4 y 4.5 muestran los resultados de ASP y *unfairness* para los distintos algoritmos de planificación sobre las plataformas Intel-2,2 y AMD-2,2.

Como se puede observar, las cargas de trabajo en el lado izquierdo de las figuras 4.4 y 4.5 experimentan mayor rendimiento que las cargas del lado derecho. La efectividad de los algoritmos de planificación reside en la capacidad de determinar el *SF* de los hilos en tiempo de ejecución. En particular, CAMP y Prop-SP (*dynamic*) pueden verse afectados por la precisión del modelo de estimación de *SF* utilizado. En la plataforma Intel, el modelo logra una estimación más precisa del *SF* que en la plataforma AMD. En Intel, la predicción de *SF* para los benchmarks de las suite SPEC CPU tienen un coeficiente de correlación de 0.97 en el core rápido y 0.96 en el core lento; mientras que en AMD, el coeficiente de correlación es 0.93 y 0.95, respectivamente. Es por esta razón que el algoritmo Prop-SP (*dynamic*) obtiene valores de *SF* peores que Prop-SP (*static*) sobre AMD. En general, los modelos de *SF* para ambas plataformas obtienen valores de *SF* bastante más precisos para aplicaciones con un *SF* alto y bajo, pero producen estimaciones menos precisas para aplicaciones con un*SF* medio. Por lo tanto, estos algoritmos son propensos a un rendimiento peor de lo esperado para cargas de trabajo que incluyen aplicaciones con *SF* medios, como 1H-1M-2L y 2M-2L.

En cuanto a los distintos algoritmos podemos hacer las siguientes observaciones:

El algoritmo HSP obtiene los mejores valores de ASP para la mayoría de las cargas de trabajo en ambas plataformas, pero a expensas de obtener los peores valores de *unfairness* en todos los casos.

CAMP obtiene un rendimiento similar a HSP, en particular para cargas de trabajo donde el número de aplicaciones con un *SF* alto coincide con el número de cores rápidos. Claramente, este no es el caso de las primeras cuatro cargas de trabajo, donde el número de aplicaciones con *SF* alto es mayor al número de cores rápidos. En este escenario CAMP debe compartir el tiempo de core rápido entre estas aplicaciones. Por esta razón, obtiene valores de *unfairness* mucho menores que HSP para estas cargas de trabajo.

Es importante mencionar cuál es el rango de *speedup factors* observados para estas aplicaciones en ambas plataformas. Sobre la plataforma Intel este rango va desde 1,47 a 2,12. En la plataforma AMD este rango es mucho más amplio y va desde 1,75 a 3,2. Debido a esto, se puede observar que en AMD-2,2 existe un incremento en la brecha de la métrica de *unfairness* entre HSP/CAMP y los restantes algoritmos de planificación.

RR y A-DWRR se comportan de manera similar en todos los casos. El hecho de compartir el tiempo de cores rápidos entre todas las aplicaciones de una carga de trabajo asegura tiempos de CPU uniformes entre estas aplicaciones. Ambos algoritmos tienen un buen desempeño en términos de ASP y *unfairness* para la mayoría de las cargas de trabajo. Sin embargo, para cargas de trabajo con aplicaciones que poseen una amplia diversidad de *SF*, tales como 3H-1L y 2H-2L, ofrecen un ASP global menor que HSP, CAMP y Prop-SP.

Nuestro algoritmo Prop-SP obtiene mejoras de rendimiento de hasta un 15 % con respecto a RR y A-DWRR, para las primeras seis cargas de trabajo. Al mismo tiempo, garantiza un valor de *unfairness* similar al de estos algoritmos. Asimismo, los valores de *unfairness* obtenidos

por Prop-SP son considerablemente inferiores a los valores obtenidos por HSP y CAMP en todos los casos.

En el caso particular de las últimas cuatro cargas de trabajo, Prop-SP obtiene valores de ASP mejores que RR y A-DWRR, pero valores de *unfairness* peores que estos algoritmos (especialmente en Intel-2,2). Esto se debe a la contención de recursos compartidos en esta plataforma. Este tipo de cargas de trabajo está compuesto por varias aplicaciones intensivas en memoria, por esta razón son más propensas a sufrir de una alta degradación en rendimiento debido a la contención de recursos compartidos [66]. Desafortunadamente, Prop-SP no tiene en cuenta el hecho que una aplicación pueda sufrir degradación debido a contención, y por lo tanto no consigue reducir la falta de justicia en estos escenarios. Sorprendentemente, RR y A-DWRR, que tampoco tienen en cuenta la contención, exhiben un comportamiento más justo para estas cargas de trabajo. Encontramos que bajo estos dos algoritmos, las aplicaciones intensivas en memoria reciben más ciclos de cores rápidos que utilizando otros algoritmos de planificación. La aceleración por el uso de core rápido compensa la degradación debido a la contención. Sin embargo, tiene un efecto negativo en el rendimiento del sistema debido a que las aplicaciones intensivas en memoria exhiben los SF más bajos en esta plataforma.

4.4.2. Aplicaciones con prioridades diferentes

Vamos a evaluar la efectividad de RR, A-DWRR y Prop-SP en escenarios donde coexisten en el sistema aplicaciones con diferentes prioridades. Para este análisis no vamos incluir los algoritmos HSP y CAMP, dado que estos algoritmos no tienen en cuenta las prioridades de las aplicaciones a la hora de tomar decisiones de planificación.

Para realizar la evaluación, se utilizó una carga de trabajo que combina aplicaciones con características muy diferentes. La carga de trabajo está compuesta por una aplicación secuencial (*h264ref*), una aplicación paralela con una porción secuencial significativa (*blast*) y una aplicación OpenMP con un alto grado de paralelismo (*fma3d*). Es importante mencionar que al utilizar otras aplicaciones con las mismas características los resultados fueron similares. La diversidad de aplicaciones presente en esta carga de trabajo nos permite explorar cómo aplicaciones diferentes pueden acelerarse a medida que se incrementan las prioridades; y cómo esto afecta el rendimiento del sistema y la justicia bajo los distintos algoritmos de planificación.

Nuestros experimentos consisten en incrementar gradualmente la prioridad determinada por el usuario (conocida como peso estático) de una aplicación de alta prioridad (HP) seleccionada, mientras se mantiene la prioridad de las aplicaciones restantes de la carga de trabajo con la configuración normal (1.0).

Para cada aplicación HP seleccionada, incrementamos la prioridad de 1 a 5. La figura 4.6 muestra los resultados obtenidos.

En el eje horizontal especificamos la aplicación HP seleccionada y su prioridad asociada

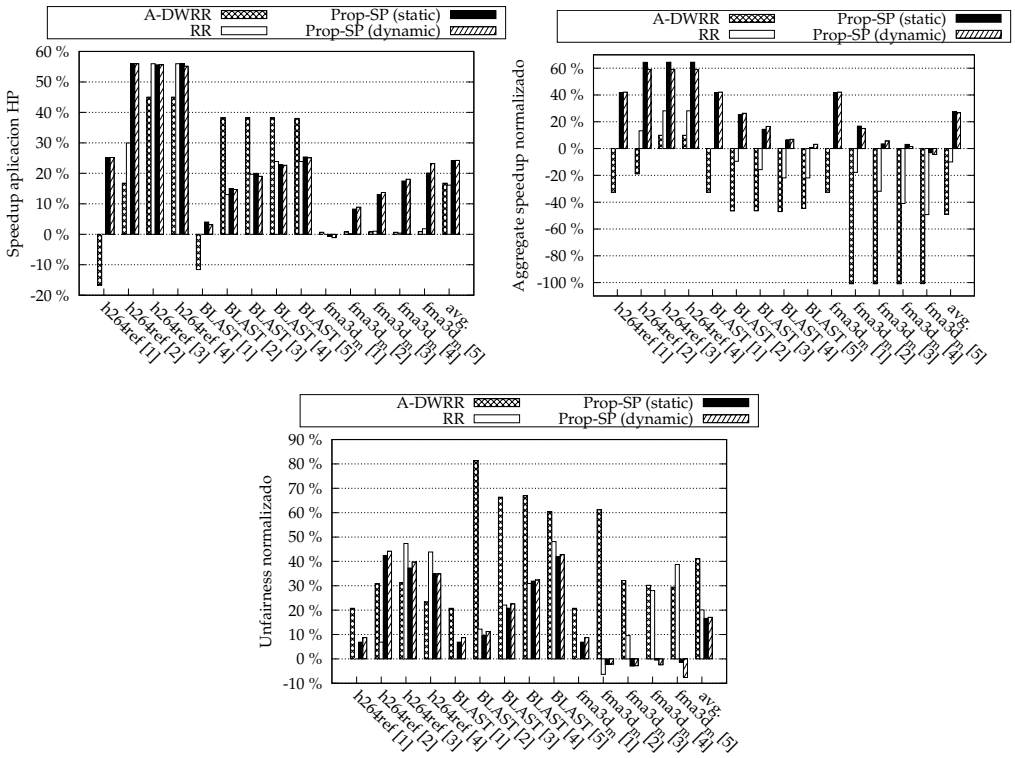


Figura 4.6: Resultados de una carga de trabajo compuesta por aplicaciones con prioridades diferentes ejecutadas sobre AMD-2,10. Las aplicaciones multi-hilo blast y fma3d se ejecutan con cinco hilos cada una.

o peso (entre paréntesis). Para las diferentes configuraciones de prioridad, mostramos los valores de ASP y *unfairness* de la carga de trabajo, así como también el *speedup* de la aplicación HP. Todas las métricas están normalizados con respecto a RR en el escenario donde todas las aplicaciones tienen la misma prioridad. Esto nos permite realizar un seguimiento de cuánto acelera la aplicación HP a medida que aumentamos su prioridad. En particular, para aplicaciones con distintas prioridades, se debe sustituir la degradación en la métrica de *unfairness* (ecuación 3.3) con su homólogo ponderado [13]. La degradación ponderada para una aplicación con peso estático w_{app} se define como:

$$WSlowdown = (Slowdown_{app} - 1) \cdot w_{app} + 1 \tag{4.11}$$

De esta forma, una aplicación con peso estático $w_{app} = 2$ la cual se degrada un 30 % (una degradación 1.3) tiene una degradación ponderada de 1.6 (60 %).

Como puede observarse, Prop-SP permite reducir el tiempo de ejecución de la aplicación HP a medida que se incrementa la prioridad, alcanzando mayor *speedup*.

RR y A-DWRR también logran una reducción relativa en el tiempo de ejecución para las

tres primeras aplicaciones a medida que aumentan las prioridades, pero no pueden hacer lo mismo con *fma3d*. Esto se debe a que *fma3d* es una aplicación OpenMP con un alto grado de paralelismo de grano fino. Al aumentar la prioridad se incrementa la porción de tiempo de core rápido que esta aplicación puede utilizar. Bajo RR y A-DWRR esta porción se distribuye de manera proporcional entre los hilos de la aplicación. Sin embargo, la distribución uniforme de la porción de tiempo de cores rápidos no contribuye a acelerar aplicaciones OpenMP altamente paralelas (ver sección 4.2.1). Por el contrario, Prop-SP utiliza AID tanto para distribuir las iteraciones de los bucles paralelos como para asignar hilos a los diferentes tipos de cores. Esto se traduce en una ganancia significativa.

Los resultados también revelan que la naturaleza de la aplicación HP determina fuertemente cómo se ve afectado el rendimiento del sistema cuando varía la prioridad. Específicamente, incrementar la prioridad de aplicaciones HP con alto *speedup* (tales como *h264ref*) conduce a un mayor rendimiento (mayores valores de ASP). En cambio, para aplicaciones HP de *speedup* medios y bajos (*blast* y *fma3d*, respectivamente), el algoritmo de planificación degrada el rendimiento a medida que aumenta la prioridad (menores valores de ASP). Esta degradación del rendimiento se hace más notoria bajo RR y A-DWRR, dado que los valores de ASP caen rápidamente a medida que aumentamos la prioridad de las aplicaciones HP con *speedup* medio y bajo. Por el contrario, Prop-SP permite mantener un rendimiento aceptable y valores menores de *unfairness* en este escenario.

4.5. Resumen del capítulo y conclusiones

En este capítulo presentamos el algoritmo de planificación Prop-SP, nuestra primera propuesta de planificación de esta tesis cuyo objetivo es mejorar el compromiso rendimiento-justicia en un AMP. Asimismo, Prop-SP proporciona soporte a prioridades de usuario y ofrece soporte para aplicaciones multi-hilo.

Para distribuir eficientemente los ciclos de cores rápidos entre las aplicaciones, Prop-SP se basa en dos mecanismos: la asignación de créditos de cores rápidos y el intercambio de hilos. Cada cierto tiempo Prop-SP asigna créditos de cores rápidos a las aplicaciones. A medida que una aplicación se ejecutan sobre un core rápido consume sus créditos. Cuando una aplicación asignada a un core rápido agota todos sus créditos, Prop-SP intentará intercambiarla con otra aplicación asignada a un core lento que posea créditos de cores rápidos.

Para aplicaciones secuenciales, el proceso de asignación de créditos asigna créditos al único hilo existente. Para aplicaciones multi-hilo, los créditos asignados a una aplicación deben distribuirse entre sus hilos en ejecución. Para esto, Prop-SP ofrece tres estrategias de distribución de créditos entre hilos: Even, BusyFC y AID. En el caso particular de AID, Prop-SP interactúa con *runtime system* a nivel de usuario. En la sección 4.2.1 demostramos que la interacción entre el SO y el *runtime system* permite obtener beneficios adicionales, especialmente en aplicaciones OpenMP regulares.

Prop-SP tiene en cuenta el beneficio relativo (*speedup*) que una aplicación obtiene al ejecutarse en cores rápidos con respecto a hacerlo en los cores lentos. Para aplicaciones secuenciales este beneficio se corresponde con el (*speedup factor*) del único hilo en ejecución. Sin embargo, el *speedup factor* no aproxima este beneficio en aplicaciones multi-hilo. Para obtener una estimación del *speedup* para una aplicación multi-hilo, es necesario tener en cuenta otros factores además del *speedup factor* de los hilos individuales. Por ejemplo: su grado de paralelismo a nivel de hilo, el número de cores rápidos de la arquitectura o cómo los créditos de cores rápidos se distribuyen entre sus hilos. En este capítulo presentamos las fórmulas para determinar el *speedup* de aplicaciones multi-hilo bajo las tres estrategias de distribución de créditos que ofrece Prop-SP.

Para determinar el *speedup factor* de un hilo en tiempo de ejecución, Prop-SP utiliza un modelo de estimación específico de la plataforma. Este modelo recolecta los valores de diversos contadores de rendimiento durante la ejecución de las aplicaciones, como por ejemplo: IPC o la tasa de fallos de último nivel de caché. La técnica para la construcción de modelos de estimación de SF utilizada en este capítulo fue propuesta en [53]. Esta técnica genera modelos de estimación precisos y sencillos para entornos AMP que difieren sólo en la frecuencia del procesador o sólo exhiben ligeras diferencias en la microarquitectura, como los usados en este capítulo.

Realizamos una evaluación experimental donde comparamos Prop-SP con otros algoritmos del estado del arte para AMPs. Implementamos estos algoritmos en el kernel de OpenSolaris y los evaluamos emulando un AMP sobre hardware real mediante la reducción en frecuencia de sus cores.

Los resultados revelan que Prop-SP es capaz de hacer un uso eficiente del AMP y mejorar el compromiso rendimiento-justicia para una amplia gama de cargas de trabajo con respecto a otros algoritmos (HSP, CAMP, RR y A-DWRR). Estos beneficios son especialmente notorios para cargas de trabajo que incluyen aplicaciones multi-hilo. Asimismo, Prop-SP es capaz de obtener valores de rendimiento y justicia aceptables cuando se tienen en cuenta prioridades de usuario.

5 Modelo analítico de rendimiento-justicia

Los resultados experimentales del capítulo anterior muestran que Prop-SP es capaz de hacer un uso eficiente del AMP y mejorar el compromiso rendimiento-justicia para una amplia gama de cargas de trabajo, con respecto a otros algoritmos. Sin embargo, desconocemos si este algoritmo es capaz de optimizar la justicia en un AMP.

En este capítulo presentamos un modelo teórico para aproximar el rendimiento global y el grado de justicia que una estrategia de planificación puede ofrecer cuando se utilizan cargas de trabajo multiprogramadas en un AMP. Empleando este modelo teórico y un simulador basado en este modelo realizamos un estudio para hallar la planificación teórica que optimiza cada una de las métricas. A partir de nuestro análisis mostramos la interrelación que existe entre el rendimiento global y la justicia en AMPs.

Este capítulo se divide en dos partes. En la primera parte derivamos las fórmulas para aproximar analíticamente las métricas de *ASP* y *unfairness*. En la segunda parte utilizamos nuestro modelo analítico para estudiar la relación ente estas dos métricas, y evaluamos la efectividad de las distintas estrategias de planificación con respecto a los planificadores que optimizan el rendimiento y la justicia en un AMP.

5.1. Derivación de las fórmulas analíticas para aggregate speedup y unfairness

Nuestro objetivo es obtener un conjunto de fórmulas para aproximar analíticamente las métricas *ASP* y *unfairness* de una carga de trabajo bajo una estrategia de planificación determinada. Para hacer más tratable el proceso de derivación hacemos los siguientes supuestos:

- Todas las aplicaciones en las cargas de trabajo consideradas en nuestro modelo tienen la misma prioridad y están compuestas por un único hilo.

- Cada aplicación exhibe un *speedup* relativo constante durante su ejecución en el AMP.

- Las fórmulas analíticas derivadas en esta sección no tienen en cuenta el *overhead* de migración y la contención por recursos compartidos.

Antes de comenzar con la derivación de las fórmulas, introducimos la siguiente notación auxiliar:

- CT_{fast}: tiempo de ejecución de la aplicación cuando se ejecuta sola sobre un core rápido.

- CT_{small}: tiempo de ejecución de la aplicación cuando se ejecuta sola sobre un core lento.

- CT_{sched}: tiempo de ejecución de la aplicación cuando se ejecuta bajo una estrategia de planificación determinada.

- N_{FC}, N_{SC}: número de cores rápidos y cores lentos del sistema AMP, respectivamente.

- SPI_{FC}, SPI_{SC}: segundos por instrucción promedio de una aplicación cuando se ejecuta en cores rápidos y lentos, respectivamente.

- SF: *speedup factor* de la aplicación secuencial donde $SF = \frac{SPI_{SC}}{SPI_{FC}}$.

- NI: número total de instrucciones que la aplicación ejecuta hasta su finalización.

- F_{fast}, F_{small}: fracción de tiempo (sobre CT_{sched}) que una aplicación ejecuta en cores rápidos y lentos, respectivamente bajo una estrategia de planificación determinada. Nótese que $F_{fast} + F_{small} = 1$. Por simplicidad, en ocasiones nos referiremos a F_{fast} como F_{app}.

- f_{fast}, f_{small}: fracción de instrucciones (sobre NI) que una aplicación completa en cores rápidos y lentos, respectivamente bajo una estrategia de planificación determinada. Nótese que $f_{fast} + f_{small} = 1$. Por simplicidad, en ocasiones nos referiremos a f_{fast} como f_{app}.

Para obtener la fórmula analítica de *unfairness* debemos calcular la degradación en rendimiento que cada aplicación experimenta bajo una estrategia de planificación determinada

($\frac{CT_{sched}}{CT_{fast}}$). A continuación derivamos la fórmula de la degradación en términos del SF y f_{fast}:

$$\begin{aligned} Slowdown &= \frac{CT_{sched}}{CT_{fast}} \\ &= \frac{NI.SPI_{FC} \cdot f_{fast} + NI \cdot SPI_{SC} \cdot f_{small}}{NI \cdot SPI_{FC}} \\ &= \frac{NI \cdot SPI_{FC} \cdot f_{fast} + NI \cdot SPI_{FC} \cdot SF \cdot (1 - f_{fast})}{NI \cdot SPI_{FC}} \\ &= \frac{NI \cdot SPI_{FC} \cdot (f_{fast} + SF \cdot (1 - f_{fast}))}{NI \cdot SPI_{FC}} \\ &= f_{fast} + SF \cdot (1 - f_{fast}) \end{aligned} \tag{5.1}$$

Derivamos ahora las fórmulas para ASP analítico bajo una estrategia de planificación determinada, en función de SF y f_{fast} para cada aplicación i:

$$\begin{aligned} ASP &= \sum_{i=1}^{n} \left(\frac{CT_{small,i}}{CT_{sched,i}} - 1 \right) \\ &= \sum_{i=1}^{n} \left(\frac{NI_i \cdot SPI_{SC,i}}{NI_i \cdot SPI_{SC,i} \cdot (\frac{f_{fast,i}}{SF_i} + f_{small,i})} - 1 \right) \\ &= \sum_{i=1}^{n} \left(\frac{1}{\frac{f_{fast,i}}{SF_i} + (1 - f_{fast,i})} - 1 \right) \end{aligned} \tag{5.2}$$

Dado que el comportamiento de los planificadores investigados se expresa en términos de la distribución de tiempo de cores rápidos que estos realizan, debemos aproximar ASP y *unfairness* en función de F_{fast} y el SF. Es importante mencionar que para una aplicación bajo un planificador determinado, obtener F_{fast} es más directo que obtener f_{fast}. Para hacer

esto posible, derivamos la fórmula que establece la relación entre F_{fast} y f_{fast}:

$$\begin{aligned} CT_{sched} &= NI \cdot SPI_{FC} \cdot f_{fast} + NI \cdot SPI_{SC} \cdot f_{small} \\ &= NI \cdot SPI_{FC} \cdot f_{fast} + NI \cdot SPI_{FC} \cdot SF \cdot (1 - f_{fast}) \\ &= NI \cdot SPI_{FC} \cdot (f_{fast} + SF \cdot (1 - f_{fast})) \end{aligned} \tag{5.3}$$

$$\begin{aligned} CT_{sched} \cdot F_{fast} \cdot \frac{1}{SPI_{FC}} &= NI \cdot f_{fast} \\ \Rightarrow CT_{sched} &= \frac{NI \cdot SPI_{FC} \cdot f_{fast}}{F_{fast}} \end{aligned} \tag{5.4}$$

Combinando las ecuaciones 5.3 y 5.4 obtenemos la siguiente expresión:

$$\begin{aligned} \frac{NI \cdot SPI_{FC} \cdot f_{fast}}{F_{fast}} &= NI \cdot SPI_{FC} \cdot (f_{fast} + SF \cdot (1 - f_{fast})) \\ \Rightarrow f_{fast} &= \frac{1}{\frac{1}{SF} \cdot \left(\frac{1}{F_{fast}} - 1\right) + 1} \end{aligned} \tag{5.5}$$

5.2. Óptimos de rendimiento y justicia

Para mostrar la interrelación entre las métricas de rendimiento y justicia, llevamos a cabo un estudio teórico que muestra la efectividad de diferentes planificadores, al ejecutar varias cargas de trabajo múltiprogramadas sintéticas sobre un AMP compuesto por dos cores rápidos ($N_{FC} = 2$) y dos cores lentos. Las cargas de trabajo empleadas en el estudio fueron construidas a partir de la combinación de cuatro aplicaciones secuenciales. En este escenario hipotético suponemos que las aplicaciones presentan ratios de rendimiento constantes entre ambos tipos de core, que oscilan entre 1.0 y 4.7, un rango de *SF* similar al observado por las aplicaciones SPEC CPU2006 ejecutadas en el prototipo asimétrico de Intel *QuickIA* utilizado en nuestros experimentos.

Cada fila en la tabla 5.1 muestra el *SF* de las aplicaciones individuales en cada carga de trabajo W_i. Para aproximar el rendimiento y la justicia, utilizamos las fórmulas analíticas derivadas en la sección 5.1 que se resumen en la tabla 5.2. Como puede observarse, las fórmulas sólo dependen de *SF* y de f_{app}, donde f_{app} representa la fracción de instrucciones de core rápido asignada por el planificador a una aplicación app durante su ejecución. Nótese que $0 \leq f_{app} \leq 1$ y $\sum_{app=1}^{n} f_{app} = N_{FC}$. Como mencionamos anteriormente, vamos aproximar

Tabla 5.1: Cargas de trabajo sintéticas

Carga	**SF$_1$**	**SF$_2$**	**SF$_3$**	**SF$_4$**
W1	4.7	4.7	1	1
W2	4.7	4.7	2.9	2.9
W3	4.3	4	4	1
W4	2.9	2.9	2.1	2.1
W5	4.7	4.7	3.6	3.6
W6	4.7	4.7	4.3	4.3
W7	3.2	2.5	1.7	1
W8	4	2.5	2.5	2.5
W9	4.7	4	3.2	2.5
W10	4	3.2	2.5	1

Tabla 5.2: Fórmulas analíticas para aproximar ASP y unfairness de una carga de trabajo, compuesta por n *aplicaciones que se ejecutan simultáneamente bajo una estrategia de planificación determinada.*

Métrica	Fórmula analítica	
ASP	$\sum_{app=1}^{n} \left(\frac{1}{\frac{f_{app}}{SF_{app}} + \left(1 - f_{app}\right)} - 1 \right)$	(5.6)
Unfairness	$\frac{MAX(Slowdown_1, \cdots, Slowdown_n)}{MIN(Slowdown_1, \cdots, Slowdown_n)}$	(5.7)
Slowdown	$f_{app} + SF_{app} \cdot \left(1 - f_{app}\right)$	(5.8)

ASP y *unfairness* en función de F_{fast} en lugar de f_{app}, por lo tanto utilizaremos la equivalencia dada por la ecuación 5.5.

La figura 5.1 muestra los valores analíticos obtenidos de *ASP* y *unfairness* para las cargas de trabajo antes mencionadas, ejecutadas bajo cuatro planificadores para AMP:

- **HSP**: investigaciones previas han demostrado que para optimizar el rendimiento en escenarios multi-aplicación el algoritmo de planificación debe seguir este enfoque [30, 54, 28]. HSP ejecuta preferentemente en cores rápidos aquellas aplicaciones que obtienen mayor beneficio de éstos. Es decir, asigna a los cores rápidos las N_{FC} aplicaciones secuenciales en la carga de trabajo que experimentan el mayor SF. Para estas aplicaciones $F_{app} = 1$; las aplicaciones restantes se asignan a los cores lentos ($F_{app} = 0$).
- **RR**: sigue una política de planificación *Round-Robin* orientada a AMPs. RR distribuye el tiempo de core rápido de forma equitativa entre las aplicaciones [8], [59]. En este escenario $F_{app} = \frac{N_{FC}}{n}$ para todas las aplicaciones.
- **Prop-SP**: en este escenario hipotético, donde las cargas de trabajo están compuestas por aplicaciones de igual prioridad, Prop-SP asigna a cada aplicación una fracción de tiempo de core rápido proporcional a su *speedup* neto ($SF_{app} - 1$).
- **Opt-unfairness**: un planificador teórico que realiza una distribución de ciclos de cores rápidos por aplicación que asegura el *unfairness* óptimo para el máximo valor de *ASP* alcanzable.

En esta tesis doctoral decimos que un algoritmo de planificación se aproxima al óptimo de justicia cuando éste obtiene valores de justicia y rendimiento global muy próximos al algoritmo teórico *Opt-Unfairness*.

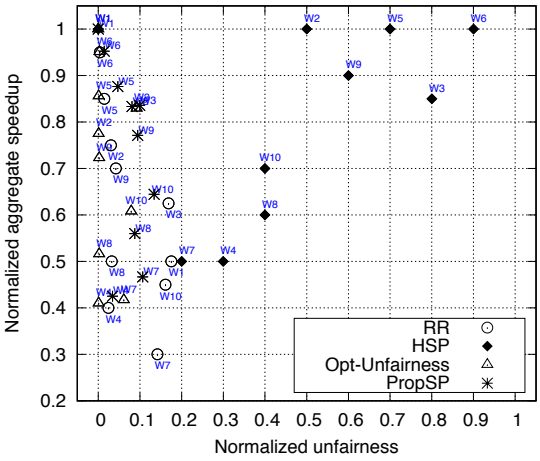


Figura 5.1: Valores de ASP y unfairness para las cargas de trabajo analizadas bajo los distintos planificadores. Ambas métricas han sido normalizadas al intervalo (0,1), donde 0 y 1 representan el mínimo y el máximo valor alcanzable en la plataforma, respectivamente. Específicamente, Normalized ASP $= \frac{ASP}{(SF_{max}-1)\cdot N_{FC}}$ *y Normalized unfairness* $= \frac{unfairness-1}{SF_{max}-1}$ *donde* SF_{max} *representa el máximo SF alcanzable en la plataforma AMP. Habrá un mejor compromiso rendimiento-justicia a medida que el valor se acerque a la esquina superior izquierda.*

Para determinar la distribución de ciclos de core rápido por aplicación bajo *Opt-unfairness* es necesario realizar una exploración exhaustiva del espacio de búsqueda. Para tal fin, creamos un simulador que hace uso de las fórmulas analíticas y busca la solución óptima para cada carga de trabajo empleando un algoritmo *branch-and-bound*. A grandes rasgos el algoritmo de búsqueda funciona de la siguiente forma. Dada una carga de trabajo, definida por el conjunto de SFs de las aplicaciones, el algoritmo calcula el par (*unfairness*, *ASP*) para cada posible distribución de los ciclos de core rápido entre las aplicaciones. Teniendo en cuenta que la exploración de todo el espacio de búsqueda continuo es inviable, las soluciones candidatas se crean variando F_{app} de 0 a 1 en pasos de 0,01, de manera que $\sum_{app=1}^{n} F_{app} = N_{FC}$. Además de esto, el algoritmo utiliza heurísticas simples para descartar las soluciones menos prometedoras.

Los resultados de la figura 5.1 muestran que HSP alcanza el máximo valor de*ASP* posible; es decir, optimiza el rendimiento global (cuanto mayor es el valor de *ASP*, mejor). Sin embargo, esto se logra a expensas de obtener los peores valores de*unfairness* (cuanto mayor es el valor de *unfairness*, peor).

Al comparar HSP con el planificador teórico *Opt-unfairness*, observamos que en la mayoría de los casos *Opt-unfairness* obtiene valores de *ASP* menores que HSP. Además, para que *Opt-unfairness* alcance el óptimo *unfairness* tiene que sacrificar significativamente el rendimiento en algunos casos (W2, W5 y W9). Este hecho pone de manifiesto que, en general, el rendimiento y la justicia son objetivos contrapuestos en AMPs y por lo tanto no es posible optimizar ambas métricas simultáneamente. Nótese que en las cargas de trabajo W3, W7 y W10, no es posible alcanzar la misma degradación en rendimiento entre todas las aplicaciones.

Por esta razón, los valores de *unfairness* normalizado bajo *Opt-unfairness* son ligeramente mayores a 0.

En cuanto a RR, los resultados ponen de manifiesto que esta política degrada tanto la justicia como el rendimiento global en comparación con *Opt-unfairness*. RR no considera el *SF* a la hora de distribuir los ciclos de core rápido entre las aplicaciones. Por esta razón, esta estrategia proporciona soluciones subóptimas en cuanto a justicia. En particular, RR sacrifica gran parte del rendimiento máximo alcanzable y en algunas cargas de trabajo, tales como W2, W7 y W10, la reducción en el rendimiento viene acompañada por degradación en la justicia.

Por último, los resultados revelan algunas buenas propiedades de Prop-SP. En primer lugar, este planificador asegura valores de *unfairness* entre un 0 y 15 % del máximo alcanzable para todas las cargas de trabajo (claramente, este no es siempre el caso para HSP y RR). En segundo lugar, Prop-SP obtiene valores de *ASP* mayores que RR en la mayoría de los casos. En tercer lugar, los valores de *ASP* que consigue Prop-SP son muy cercanos al máximo alcanzable dado por HSP en muchos casos.

A pesar de las buenas propiedades de Prop-SP, este planificador aún degrada la justicia con respecto al óptimo (*Opt-unfairness*) para gran parte de las cargas de trabajo consideradas. Además, en cargas de trabajo como W2 y W5 viene acompañado de una importante degradación en el rendimiento. Esto pone de manifiesto que, a pesar de sus beneficios, Prop-SP dista considerablemente del planificador óptimo de justicia. A partir del modelo teórico derivado en este capítulo y al simulador basado en él, fue posible diseñar una mejor estrategia: el planificador ACFS que presentamos en el siguiente capítulo.

5.3. Resumen del capítulo y conclusiones

En este capítulo presentamos un modelo teórico para analizar la interrelación entre el rendimiento global y la justicia en un AMP. Empleando este modelo teórico y un simulador basado en él, llevamos a cabo un estudio que nos permitió mostrar la efectividad de diferentes estrategias de planificación al ejecutar varias cargas de trabajo múltiprogramadas sintéticas sobre un AMP.

Investigaciones previas han demostrado que para optimizar el rendimiento global el algoritmo de planificación debe seguir el enfoque *High-SPeedup* (HSP) [30, 54, 28], es decir ejecutar preferentemente en cores rápidos aquellas aplicaciones que obtienen un mayor beneficio relativo (*speedup*) de estos. Los resultados de nuestro estudio revelan que HSP optimiza el rendimiento a expensas de degradar la justicia.

Por otro lado, analizamos *Opt-unfairness*, un planificador teórico que optimiza la justicia. Este planificador asegura un valor óptimo de *unfairness* para el máximo rendimiento global alcanzable. Como mostramos en nuestro estudio, para alcanzar el óptimo *unfairness* es preciso sacrificar significativamente el rendimiento en algunos casos. Este hecho pone de

manifiesto que el rendimiento y la justicia son objetivos de optimización contrapuestos en AMPs.

En nuestro estudio teórico también analizamos la efectividad del planificador RR, un algoritmo que reparte los ciclos de core rápido de forma equitativa entre aplicaciones para mejorar la justicia en el AMP. Los resultados revelan que esta política constituye una solución subóptima, ya que degrada tanto la justicia como el rendimiento en comparación con *Opt-unfairness*. Esto se debe principalmente a que RR no tiene en cuenta el *speedup* relativo a la hora de distribuir los ciclos de core rápido entre las aplicaciones.

Finalmente, nuestro estudio revela que el planificador Prop-SP ofrece un mejor compromiso entre rendimiento y justicia que RR. Sin embargo, Prop-SP también constituye una estrategia subóptima de justicia, que dista considerablemente del planificador teórico *Opt-unfairness* en algunos escenarios. Por lo tanto, fue necesario continuar en la búsqueda de una mejor estrategia con el objetivo de optimizar la justicia en un AMP.

6 Algoritmo de planificación ACFS

El análisis teórico del capítulo anterior nos permitió determinar la planificación (reparto de ciclos de cores rápidos y lentos entre aplicaciones) que optimiza la justicia para el valor máximo de rendimiento global alcanzable en escenarios sintéticos. Nuestro análisis revela que la estrategia de reparto de ciclos de core rápido realizada por Prop-SP, que por su simplicidad puede implementarse eficientemente en un sistema operativo real, permite mejorar el compromiso rendimiento-justicia. Sin embargo, Prop-SP se aleja considerablemente del óptimo.

La complejidad exponencial del algoritmo exhaustivo de búsqueda del óptimo empleado en nuestro simulador imposibilita su inclusión en el kernel del SO. Esto dificulta la aproximación del reparto de ciclos óptimo en cuanto a justicia desde el planificador del sistema. Para mitigar este problema, diseñamos el algoritmo de planificación *Asymmetry-aware Completely Fair Scheduler* (ACFS), que intenta aproximar el planificador óptimo de justicia mediante el seguimiento del progreso relativo realizado por los distintos hilos de la carga de trabajo, a medida que éstos se ejecutan en los distintos tipos de cores del AMP.

Además de haber sido diseñado para optimizar la justicia en un AMP, ACFS está equipado con un parámetro de configuración, denominado *Unfairness Factor (UF)*, que permite incrementar el rendimiento global en escenarios con requisitos de justicia menos exigentes. Es decir, este parámetro posibilita que el planificador mejore el rendimiento global gradualmente a costa de degradar la justicia. A pesar de las diferencias entre Prop-SP y ACFS, nuestra nueva propuesta incorpora de forma efectiva el soporte para aplicaciones multihilo de Prop-SP, que permite repartir de distintas formas los ciclos de core rápido disponibles entre los hilos de una misma aplicación.

Para la evaluación de ACFS empleamos el prototipo QuickIA de Intel [1]. Esta fue la primera experiencia con hardware asimétrico real de la tesis que nos permitió identificar un importan-

[1] Esto pudo llevarse a cabo gracias a la donación del prototipo por parte del Operating System Group in the Circuits and Systems Research Lab de Intel en Hillsboro, Oregon (USA).

te desafío: aproximar de forma precisa el *speedup factor* de un hilo en tiempo de ejecución sobre una plataforma AMP real, que combina cores con ejecución en orden y fuera de orden. En particular, detectamos que las técnicas propuestas previamente para aproximar el *speedup factor* sobre multicore asimétricos con diferencias menos pronunciadas entre los cores (como la reducción en frecuencia) [53, 28] no permiten obtener *speedups* suficientemente precisos en el QuickIA de Intel. Para superar esta limitación, en este capítulo proponemos una metodología basada en análisis de fases y en el uso de contadores hardware, que permite construir modelos de estimación de *speedup factor* precisos para sistemas donde los cores presentan diferencias significativas a nivel microarquitectónico. Nótese además que a diferencia de otras estrategias [48, 60], los modelos generados con nuestra metodología no requieren insertar información en los archivos ejecutables de las aplicaciones.

Comenzamos este capítulo describiendo el diseño del algoritmo ACFS. Continuamos con una descripción de la metodología utilizada para derivar modelos de estimación de *SF* sobre multicore asimétricos. Para finalizar, llevamos a cabo un análisis experimental donde comparamos la efectividad de ACFS con la de otros algoritmos del estado del arte para AMPs. Asimismo, mostramos la eficacia del parámetro de configuración *Unfairness Factor* de ACFS.

6.1. Diseño del planificador ACFS

En esta sección comenzamos describiendo el mecanismo utilizado por ACFS para realizar el seguimiento del progreso de los hilos de ejecución, así como la asignación inicial de hilos a cores que realiza el planificador. A continuación, analizamos en detalle un ejemplo que ilustra el funcionamiento del mecanismo de seguimiento del progreso de un hilo. Este ejemplo pone de manifiesto que en ocasiones es necesario realizar intercambios de hilos para asegurar que las aplicaciones en ejecución experimenten una degradación en rendimiento (*slowdown*) similar, lo cual es necesario para garantizar justicia en el sistema. Para finalizar, describimos el funcionamiento del parámetro de configuración *unfairness_factor*, que permite al administrador del sistema ajustar el compromiso entre la justicia y el rendimiento global en el AMP.

6.1.1. Seguimiento del progreso y asignación inicial de hilos a cores

Para llevar la cuenta del progreso realizado por las distintas aplicaciones en el sistema, el planificador asocia a cada hilo un contador de progreso llamado *amp_vruntime*. Cuando un hilo se ejecuta durante un *tick* de reloj en un tipo de core determinado, ACFS incrementa el contador *amp_vruntime* del hilo en $\Delta amp\_vruntime$, que se calcula como sigue:

$$\Delta amp\_vruntime = \frac{100 \cdot W_{def}}{S_{core} \cdot W_t} \tag{6.1}$$

Donde W_{def} es el peso de las aplicaciones con la prioridad por defecto[2], S_{core} es la degradación en rendimiento (*slowdown*) experimentada por la aplicación durante ese tick de reloj (relativo a usar los cores rápidos en el AMP) y W_t es el peso del hilo, valor que se extrae directamente a partir de la prioridad de la aplicación especificada por el usuario.

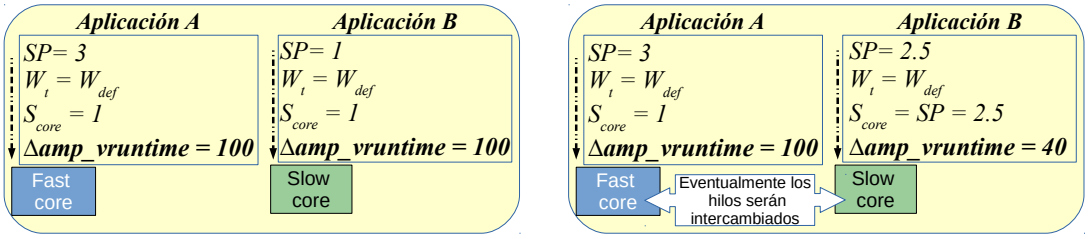
Cuando un hilo se ejecuta en un core rápido, $S_{core} = 1$ (no hay degradación del rendimiento). Por el contrario, si el hilo se ejecuta en un core lento $S_{core} = SP$, donde SP representa el *speedup* que alcanza la aplicación de este hilo al utilizar los cores rápidos del AMP con respecto a usar solamente los cores lentos. Notése que SP puede variar a lo largo del tiempo durante las diferentes fases que atraviesa un programa, y además estas variaciones de *speedup* deben ser consideradas por el planificador para realizar un seguimiento preciso del progreso relativo de una aplicación a lo largo del tiempo.

Al igual que Prop-SP, ACFS aproxima el valor de SP en tiempo de ejecución teniendo en cuenta el SF del hilo y el número de hilos activos de la aplicación (una aproximación del grado de paralelismo a nivel de hilo). A su vez, el SF del hilo se estima alimentando un modelo de estimación con valores de métricas de rendimiento (tales como IPC o la tasa de fallos del último nivel de cache) obtenidos para el hilo mediante contadores hardware. En la sección 6.3.1 proporcionamos más información sobre el mecanismo de estimación del SF.

Para aplicaciones secuenciales, el contador *amp_vruntime* asociado al único hilo en ejecución representa el progreso que la aplicación ha realizado hasta el momento con respecto al progreso que habría logrado si se hubiera ejecutado completamente sobre un core rápido. Esta idea se ilustra en el ejemplo analizado en la siguiente sección. Para aplicaciones multihilo, monitorizar el progreso de forma precisa resulta mucho más complejo, porque dos o más hilos de una misma aplicación pueden asignarse a diferentes tipos de cores simultáneamente, y los hilos pueden llegar a bloquearse por motivos de sincronización durante cortos períodos de tiempo. Para evitar mantener un contador de progreso global por aplicación, lo que podría causar contención por la actualización desde distintas CPUs simultáneamente, ACFS utiliza los contadores *amp_vruntime* de los distintos hilos de la aplicación para asegurar justicia. No obstante, tal como revelan nuestros experimentos (sección 6.4), utilizar los contadores *amp_vruntime* por hilo permite al planificador proporcionar justicia en el sistema en escenarios multi-aplicación. Nótese además que mantener los contadores de progreso por hilo también permite a ACFS controlar el progreso realizado por un hilo de una aplicación específica con respecto al del resto de hilos de la misma aplicación. Investigaciones previas [26, 12, 27] han demostrado que para lograr un buen rendimiento en aplicaciones HPC escalables es importante garantizar que todos sus hilos realicen el mismo progreso.

Cuando un nuevo hilo ingresa al sistema, ACFS lo asigna al core menos ocupado de la plataforma para mantener la carga balanceada. Al hacer esto, ACFS intenta asignar preferentemente los cores rápidos, ya que esto contribuye a maximizar el rendimiento del sistema [35].

[2] La estrategia utilizada por ACFS para tener en cuenta las prioridades durante el *CPU accounting* está inspirada en el mecanismo utilizado por el planificador CFS de Linux 2.2.2.5



(a) Escenario 1 (b) Escenario 2

Figura 6.1: Asignación hipotética de hilos a cores bajo el algoritmos de planificación ACFS.

Para un hilo recién creado, el valor del contador *amp_vruntime* se establece al valor máximo observado de *amp_vruntime* entre todos los hilos en el sistema. Este valor inicial permite una comparación justa entre los *amp_vruntime* de los hilos que ingresaron al sistema en instantes diferentes.

6.1.2. Caso de estudio

Para ilustrar la idea que hay tras el mecanismo de seguimiento del progreso empleado por ACFS, consideraremos el escenario hipotético que se muestra la figura 6.1a. Supongamos que existen dos aplicaciones secuenciales (A y B) con prioridad por defecto ($W_{def} = W_t$), que se ejecutan en un AMP compuesto por un core rápido y un core lento. Inicialmente el planificador ACFS asigna una aplicación a cada core para mantener la carga balanceada. Por ejemplo, supongamos que la aplicación A se asigna al core rápido y la aplicación B al lento. Como se muestra en la figura 6.1a, la aplicación A está ejecutando una fase del programa donde $SP = 3$, lo que implica que el rendimiento en el core rápido es tres veces mayor que en un core lento. Por el contrario, la aplicación B está actualmente ejecutando una fase del programa donde no se obtiene ningún beneficio al ejecutarse en un core rápido frente a uno lento ($SP = 1$). En estas circunstancias, el contador *amp_vruntime* de ambas aplicaciones se incrementa en 100 unidades por *tick* de acuerdo a la ecuación 6.1. Este incremento refleja que las dos aplicaciones realizan el mismo progreso, que además es el máximo progreso alcanzable (100 %). En este caso se consigue el valor óptimo de injusticia, ya que ambas aplicaciones no experimentan ninguna degradación con respecto a cuando se ejecutan solas, y también el máximo rendimiento global alcanzable, la aplicación con el *speedup* relativo más alto se asigna al core rápido. En este escenario, ACFS actúa de manera óptima, es decir, como el planificador teórico *Opt-Unfairness* presentado en el capítulo anterior.

Supongamos ahora que la aplicación B comienza a ejecutar una nueva fase del programa donde $SP = 2,5$, como se muestra en la figura 6.1b. En este escenario, el rendimiento de la aplicación B sí podría incrementarse asignando la aplicación al core rápido. Aunque mantener la asignación actual de hilos a cores seguiría proporcionando el máximo rendimiento global, debido a que la aplicación con mayor *speedup* (la aplicación A) aún está corriendo en el core

rápido, esta asignación no garantiza justicia. Intuitivamente, el contador *amp_vruntime* de la aplicación B se incrementará en 40 unidades por cada *tick*, lo cual indica que B alcanza solamente el 40 % del progreso máximo teórico. Por el contrario, el contador *amp_vruntime* de la aplicación A se incrementa en 100 unidades por cada *tick*. Si se mantiene esta asignación de aplicaciones a cores, la diferencia entre los contadores *amp_vruntime* se incrementará a lo largo del tiempo, lo cual se traducirá en injusticia (distinto progreso relativo).

6.1.3. Garantizando justicia mediante el intercambio de hilos

Para garantizar justicia, el planificador debe lograr que las aplicaciones realicen el mismo progreso. Para ello, ACFS intenta garantizar que el valor de los contadores de progreso de los hilos, sea lo más cercano posible. Esto requiere realizar migraciones de hilos entre los diferentes tipos de core cada cierto tiempo. Cabe destacar que, en general, es preferible realizar *intercambios de hilos* en vez de migraciones de hilos individuales en un único sentido, dado que los intercambios contribuyen a garantizar el balance de carga.

Dado que las migraciones frecuentes pueden introducir *overhead* en la ejecución, ACFS no efectúa intercambios tan pronto como detecte que un hilo T_A ejecutándose en un core rápido posea mayor *amp_vruntime* que el que posee otro hilo T_B ejecutándose en un core lento. Por el contrario, ACFS sólo intercambiará dos hilos T_A y T_B si se cumple que la diferencia entre los contadores de progreso ($amp\_vruntime(T_A) - amp\_vruntime(T_B)$) supera un cierto umbral.

En general mantener un valor alto para el umbral, permite garantizar una frecuencia de intercambio de hilos reducida, y con ello mitigar el *overhead* de las migraciones. Sin embargo, el uso de valores altos para el umbral también contribuye a "acumular injusticia" en el sistema durante largos períodos de tiempo.

Cabe destacar que, cuando el número de hilos activos de la carga varía, mantener un valor fijo del umbral no garantiza que los intercambios de hilos se produzcan a una frecuencia media determinada. Concretamente, si T_{swap} es el período de intercambio de hilos medio registrado durante un cierto tiempo *Total_time*. Definimos este período como sigue:

$$T_{swap} = \frac{Total\_time * N_{FC}}{Total\_swaps} \tag{6.2}$$

donde N_{FC} es el número de cores rápidos en el AMP, y *Total_swaps* representa el número de intercambios de hilos registrados durante el intervalo de tiempo *Total_time*. Para garantizar que el planificador realiza los intercambios de hilos a una frecuencia controlada (valor dado para T_{swap}), el umbral de intercambio de hilos debe ajustarse apropiadamente teniendo en cuenta la carga del sistema y el número de cores rápidos de la plataforma.

Delegar al administrador del sistema la responsabilidad de establecer manualmente este umbral puede resultar una tarea tediosa, y poco práctica, especialmente cuando se ejecutan cargas de trabajo donde el número de hilos activos varía frecuentemente a lo largo de la

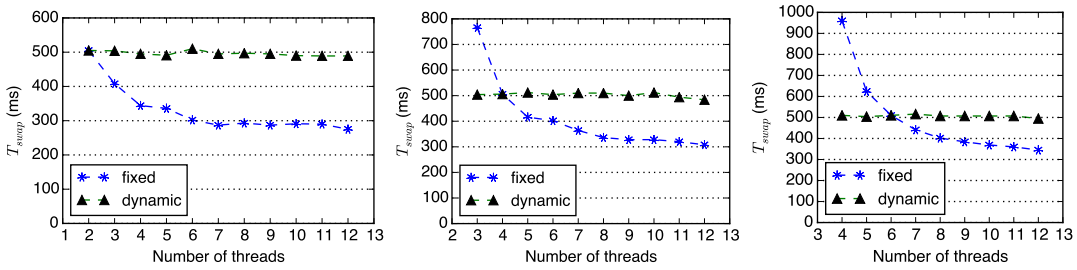
ejecución. Para superar esta limitación, ACFS posee un parámetro de configuración, llamado *amp_threshold*, que representa un valor *base* para el umbral. Este valor base permite al usuario establecer un valor específico de T_{swap} cuando el número total de hilos es el doble que el número de cores rápidos. En tiempo de ejecución, ACFS utiliza un umbral interno que se establece automáticamente multiplicando $amp\_threshold$ por el siguiente factor de escalado[3]: $\frac{2\cdot(N_T-N_{FC})}{N_T}$, donde N_T es el número total de hilos. Intuitivamente, cuando el número de hilos no excede el número de cores rápidos, ACFS no realiza migraciones; el factor de escalado no se tiene en cuenta en este caso. En la práctica, este factor garantiza que el umbral interno usado por ACFS es menor que $amp\_threshold$ cuando $N_{FC} < N_T < 2 \cdot N_{FC}$. Por el contrario, cuando $N_T > 2 \cdot N_{FC}$ el umbral interno es mayor que $amp\_threshold$. Para alcanzar un período de intercambio promedio (T_{swap}) deseado, el valor base del umbral *amp_threshold* puede calcularse empleando la siguiente ecuación:

$$amp\_threshold = 100 \cdot \frac{T_{swap}}{2 \cdot T_{tick}} \cdot (1 - \frac{1}{SF_{avg}}) \tag{6.3}$$

donde SF_{avg} es el *speedup factor* promedio observado en la plataforma y T_{tick} es el período de un *tick* del planificador. Notesé que para derivar la ecuación 6.3 analíticamente, calculamos el tiempo transcurrido para que dos aplicaciones secuenciales (con $W_t = W_{def}$) que acaban de ser intercambiadas, sean intercambiadas de nuevo en un escenario con $N_T = 2 \cdot N_{FC}$.

Para demostrar que el ajuste dinámico del umbral de intercambio que realiza ACFS garantiza un período de intercambio específico, realizamos un estudio de sensibilidad usando distintas configuraciones AMP sobre la plataforma AMD de 12 cores descrita en la sección 2.1. En particular, nuestro experimento consiste en ejecutar un determinado número de instancias (que varían desde $N_{FC} + 1$ al número total de cores) de una misma aplicación secuencial sobre cada configuración AMP. Para todas las ejecuciones establecimos un valor constante de $amp\_threshold$ para forzar un período de intercambio promedio de $500ms$.

La figura 6.2 muestra como varía el período promedio de intercambio de hilos real (T_{swap}) para diferente número de hilos al emplear un umbral dinámico, y uno fijo (igual a *amp_threshold*). Como se puede observar en la figura, ajustar el umbral dinámicamente permite a ACFS aproximar el período de intercambio promedio deseado, mientras que usar un umbral constante solo garantiza el período específico en el caso base ($N_T = 2 \cdot N_{FC}$). En la práctica, el factor de escalado utilizado actualiza el umbral de forma efectiva a medida que el número de hilos aumenta, lo que permite aproximar el período de intercambio deseado.



(a) 1F-11S (b) 2F-10S (c) 3F-9S

Figura 6.2: T_{swap} para diferente número de hilos en varias configuraciones AMP.

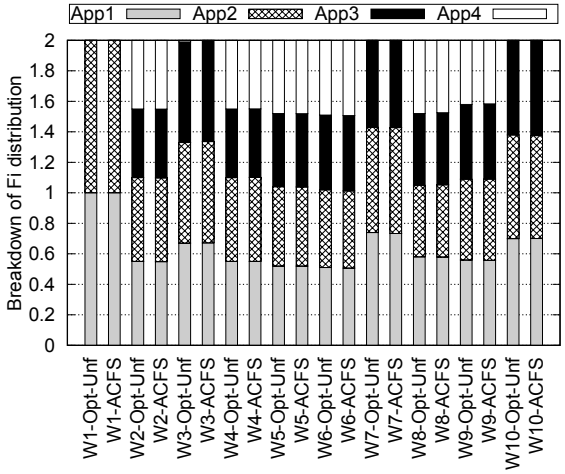


Figura 6.3: Distribución de tiempo de core rápido (F_i) entre las aplicaciones de las cargas de trabajo en la tabla 5.1 bajo Opt-Unfairness y ACFS.

6.1.4. Compromiso rendimiento-justicia

Antes de llevar a cabo la implementación y evaluación de ACFS en un sistema real, analizamos nuestra propuesta de planificación sobre el simulador descrito en el capítulo 5, para realizar una comparación con el planificador óptimo de justicia *Opt-unfairness*. Nuestro análisis revela que en el escenario sintético considerado en el capítulo anterior y asumiendo estimaciones de *speedup* perfectas, el diseño base de ACFS (descrito en la sección 6.1) se comporta de forma muy similar a *Opt-unfairness*: ACFS consigue valores de justicia y rendimiento global comprendidos en un rango inferior al 1 % con respecto a los valores de *Opt-unfairness*. Esto se debe a que ambos planificadores realizan casi la misma distribución de ciclos de core rápido entre aplicaciones. La figura 6.3 muestra las similitudes entre la distribución de ciclos de core rápido realizada por ambos planificadores para las cargas de trabajo sintéticas consideradas en el estudio teórico del capítulo anterior (tabla 5.1). Es importante mencionar

[3] Este factor fue determinado empíricamente analizando cómo varia el período de intercambio promedio con el número de hilos, pero mantenido un valor de umbral constante bajo diferentes configuraciones AMP. En escenarios donde el número de hilos supera al número de cores (*oversubscription*), el factor se reduce teniendo en cuenta también la longitud media de las *run queues* del planificador.

que observamos las mismas tendencias con todas las cargas de trabajo sintéticas adicionales con las que experimentamos.

Dado que la justicia y el rendimiento global son objetivos contrapuestos (como mostramos en el estudio teórico del capítulo 5), *Opt-Unfairness* podría degradar el rendimiento del sistema significativamente a expensas de proporcionar el valor óptimo de *unfairness*. Para superar esta limitación el planificador ACFS está equipado con un parámetro de configuración llamado *Unfairness Factor (UF)*, que permite al administrador del sistema ajustar la relación entre la justicia y el rendimiento global en escenarios donde los requerimientos de justicia no son tan estrictos. Cuando $UF = 1{,}0$ (valor por defecto) el planificador se comporta como la implementación base, es decir intenta obtener el máximo rendimiento global alcanzable para el valor óptimo de *unfairness*. Para valores mayores de *UF*, el planificador incrementa el rendimiento a costa de degradar la justicia hasta cierto punto. Intuitivamente, esto puede lograrse incrementando la fracción de tiempo de core rápido que se otorga a aquellas aplicaciones de la carga de trabajo que tienen un *speedup* más alto, mientras se reduce la fracción de tiempo de core rápido asignada al resto de aplicaciones. El principal desafío está en cómo incrementar gradualmente el rendimiento mientras la justicia se mantiene en niveles aceptables. Para lograr este objetivo, es necesario tener en cuenta el parámetro *UF* a la hora de actualizar el contador de progreso *amp_vruntime* de un hilo cada *tick*. En particular, esto se consigue reemplazando la prioridad o peso estático del hilo (W_t) en la ecuación 6.1 por su peso dinámico (DW_t), que se define como sigue:

$$DW_t = W_t \cdot \left(1 + \frac{(UF-1)\cdot(SP - S_{min})}{S_{max} - S_{min}}\right) \tag{6.4}$$

donde S_{max} y S_{min} es el *speedup* (SPs) máximo y mínimo (respectivamente) observado entre todas las aplicaciones de la carga de trabajo.

Esencialmente, reemplazando el peso estatico del hilo (W_t) por su peso dinámico (DW_t), el *amp_vruntime* de los hilos con *speedup factor* alto se incrementa a un ritmo más lento que el de los hilos con un *speedup factor* más bajo. Esto se traduce en una fracción de tiempo de core rápido mayor para aplicaciones con *speedup* alto, lo que contribuye a incrementar el rendimiento global.

Utilizando nuestra implementación de ACFS en el simulador descrito en el capítulo 5, observamos que incrementar gradualmente el *UF* para una carga de trabajo sintética se traduce en un incremento gradual del rendimiento global bajo ACFS. Al mismo tiempo el planificador garantiza un valor de *unfairness* menor o igual que $UF \cdot opt$ donde *opt* representa el *unfairness* óptimo que puede obtenerse para la carga de trabajo en cuestión. Para ilustrar esta tendencia, la figura 6.4 muestra los valores analíticos de *ASP* y *unfairness* para algunas de las cargas de trabajo sintéticas empleadas en el capítulo anterior (tabla 5.1). Los diferentes valores de *UF* empleados en nuestro análisis, comprendidos entre 1,0 y 1,5; se muestran junto a cada punto representado en la figura. Cabe destacar que estos resultados sólo pueden obtenerse

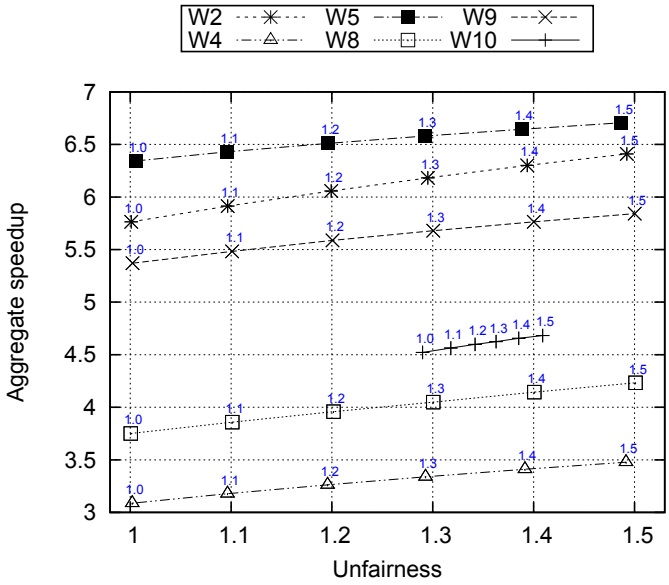


Figura 6.4: ASP y Unfairness teóricos para los diferentes valores de UF

con estimaciones de *speedup* perfectas. En la sección 6.4.3 analizamos la efectividad del parámetro *UF* usando nuestra implementación de ACFS sobre el kernel Linux, y empleando una plataforma asimétrica real.

6.2. Soporte para aplicaciones multi-hilo en ACFS

En el capítulo 4 describimos cómo el algoritmo Prop-SP distribuye el tiempo de uso de cores rápidos entre las aplicaciones. En particular, Prop-SP realiza esta distribución asignando créditos de core rápido a cada aplicación. En el caso particular de una aplicación multi-hilo, es necesario distribuir esos créditos entre los hilos activos de la aplicación. Para proporcionar soporte para distintos tipos de aplicaciones multi-hilo, Prop-SP soporta tres estrategias de distribución de créditos: BusyFC, Even y AID. Como se menciona en el capítulo 4 la última estrategia de distribución (AID) requiere la interacción entre el *runtime system* y el SO.

A diferencia de Prop-SP, ACFS no emplea créditos, sino que utiliza un contador de progreso por cada hilo (*amp_vruntime*). A pesar de las diferencias entre ambos planificadores, es posible adaptar de forma sencilla las estrategias de distribución de créditos de Prop-SP, para su uso en ACFS. Concretamente, bajo ACFS, los mecanismos Even, BusyFC y AID funcionan de la siguiente manera:

- **Even** distribuye el tiempo de uso de cores rápidos de manera uniforme entre todos los hilos de la aplicación. Para lograr esta distribución de tiempo bajo ACFS no es necesario hacer ninguna modificación en el planificador, dado que el mecanismo para garantizar justicia de ACFS ya intenta que todos los hilos de una misma aplicación realicen el mismo progreso.

- **BusyFCs** tiende a favorecer a aquellos hilos que están actualmente asignados a cores rápidos. Para implementar este mecanismo sobre ACFS se realizan intercambios de los contadores de progreso (*amp_vruntime*) entre hilos de una misma aplicación. De esta forma se evita realizar muchas migraciones que pueden afectar negativamente al rendimiento en algunos casos. Por ejemplo, supongamos que un hilo T_{fast} se encuentra en ejecución sobre un core rápido y otro hilo T_{slow} se encuentra asignado a un core lento. Supongamos también que sus contadores de progreso se denotan mediante C_f y C_s, respectivamente. Tarde o temprano, el contador C_f será mayor que el contador C_s, por la asignación de hilos a cores. En este escenario, ACFS debería intercambiar ambos hilos. Sin embargo, si los hilos pertenecen a la misma aplicación y se selecciona una estrategia BusyFC, el planificador intercambiará únicamente el valor de los contadores de progreso entre ambos hilos.

- **AID** interactúa con el *runtime system* a nivel de usuario para lograr un balance de carga adecuado en la distribución de las iteraciones de bucles paralelos. Al mismo tiempo, utiliza un enfoque BusyFCs para favorecer hilos con menor identificador de hilo *TID*. Específicamente, al seleccionar la estrategia AID para una aplicación, ACFS solo intercambiará dos hilos de la misma aplicación entre cores si el hilo que se ejecuta en el core rápido tiene mayor *TID* que el que que se ejecuta sobre el lento. Adicionalmente, cuando la diferencia entre los contadores de progreso de dos hilos asignados a distintos tipos de cores supera el umbral de intercambio de ACFS, pero los hilos están asignados legítimamente al tipo de core que les corresponde por *TID*, AID se comporta de manera similar a BusyFC intercambiando sólo los contadores de los hilos.

Por último, recordemos que el *speedup* de una aplicación secuencial se corresponde con el *speedup factor* del único hilo en ejecución. Sin embargo, para calcular el *speedup* de una aplicación multi-hilo es necesario tener en cuenta otros factores, además del *speedup factor* de los hilos. El *speedup* de estas aplicaciones bajo ACFS dependerá de la estrategia de distribución de ciclos de core rápido utilizada (BusyFC, Even o AID). En la tabla 4.1 de la sección 4.2.2 presentamos las fórmulas para aproximar el *speedup* de las aplicaciones multi-hilo bajo las distintas estrategias. ACFS considera el *speedup* de la aplicación como un todo a la hora de actualizar los contadores de progreso de los hilos.

6.3. Determinando el speedup factor en tiempo de ejecución

Como mencionamos en la sección anterior, el planificador ACFS tiene en cuenta el *SF* de un hilo a la hora de actualizar su contador de progreso durante la ejecución. Para determinar el *SF* del hilo en cualquier instante, ACFS alimenta un modelo de estimación específico de la plataforma con valores de distintas métricas de rendimiento recolectadas a lo largo del tiempo. Dado que el valor de una métrica de rendimiento (por ejemplo, IPC) puede diferir entre cores de distinto tipo, el planificador emplea dos modelos: uno para predecir el *SF* a partir de métricas obtenidas en el core rápido y otra para predecirlo a partir de métricas obtenidas en

el core lento. Estos dos modelos de estimación (usados para todos los hilos) se generan *offline*. Es importante destacar que la construcción de modelos de estimación precisos constituye un importante desafío, ya que requiere la identificación de un subconjunto reducido de métricas de rendimiento que permitan explicar las diferencias que surgen al ejecutar un hilo en distintos tipos de core [28, 53].

El resto de esta sección se organiza de la siguiente manera. La sección 6.3.1 introduce los detalles acerca de las plataformas experimentales utilizadas, que son cruciales para entender los desafíos que surgen al construir modelos de estimación de *SF* en distintos sistemas. La sección 6.3.2 presenta nuestra metodología para generar estos modelos mediante procesamiento *offline*. Para finalizar, la sección 6.3.3 describe el modo en el que nuestra implementación del planificador ACFS en el kernel Linux utiliza los modelos de estimación en tiempo de ejecución.

6.3.1. Plataforma experimental

Para evaluar la efectividad de ACFS se utilizó el prototipo asimétrico QuickIA de Intel [10] y la plataforma AMD-12. La descripción detallada de ambas plataformas puede encontrarse en la sección 2.1.

El prototipo QuickIA de Intel integra en un mismo sistema dos procesadores multicore: un Intel Xeon E5450 (quad core) y un Intel Atom N330 (dual core). Nótese que el procesador Intel Xeon integra dos grupos de dos cores cada uno, con una cache de último nivel(L2) compartida por cada grupo de cores. Para reducir el efecto de la contención por recursos compartidos en los experimentos sobre esta plataforma[4], deshabilitamos un core de cada grupo en el procesador Xeon. De este modo, nuestra configuración asimétrica esta compuesta de dos cores rápidos (Xeon E5450) y dos cores lentos (Atom N330). Nos referiremos a esta configuración como 2F-2S.

Los diferentes cores en esta plataforma cuentan con tres contadores hardware de propósito específico y dos contadores hardware de propósito general (configurables). En la práctica, esto permite monitorizar tres métricas de rendimiento simultáneamente en cualquier tipo de core: el IPC, usando los contadores de propósito específico, y otras dos métricas de alto nivel usando los contadores de propósito general. Esta restricción en el número de contadores hardware, sumado a las diferencias profundas entre cores hace que la estimación del *SF* en esta plataforma sea una tarea compleja.

El escaso número de cores en el prototipo Intel QuickIA hace que no resulte una plataforma adecuada para evaluar cargas de trabajo que incluyen aplicaciones multi-hilo. Por esta razón experimentamos también con la plataforma AMD-12. Sobre esta plataforma emulamos un sistema AMP compuesto por 2 cores rápidos y 10 cores lentos (2F-10S), mediante la reducción

[4] Nótese que ninguno de los algoritmos de planificación analizados en esta tesis considera los efectos asociados a la contención de recursos compartidos. Hacer consciente a un planificador *asymmetry aware* de estos efectos constituye una interesante línea de trabajo futuro.

de la frecuencia del procesador en algunos cores. En particular, la frecuencia de los cores rápidos en 2F-10S es 2,1 GHz, mientras que la frecuencia de los cores lentos es 800 MHz. Cada core en esta plataforma está equipado con cuatro contadores hardware configurables.

6.3.2. Generando modelos de estimación de speedup factor *offline*

Para construir modelos de estimación de *speedup factor* para las plataformas descritas en la sección anterior utilizamos inicialmente la metodología presentada en [53]. Básicamente, para aplicar esta metodología se deben seguir los siguientes pasos:

1. Seleccionar un conjunto representativo AP de aplicaciones secuenciales y un conjunto de métricas de rendimiento M que permitan caracterizar el comportamiento de las aplicaciones a nivel microarquitectónico y a nivel de la jerarquía de memoria.

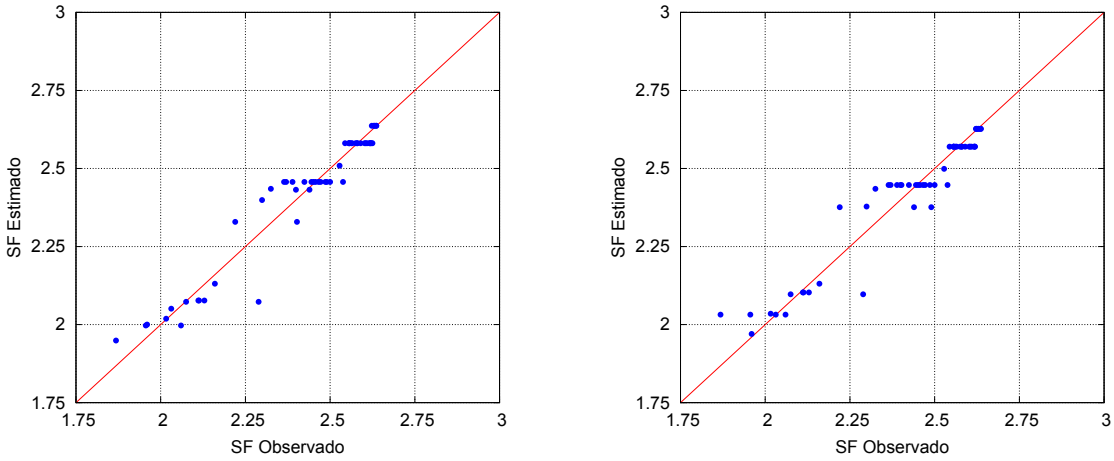
2. Ejecutar las aplicaciones del conjunto AP en ambos tipos de core para obtener su *speedup factor medio* (es decir, el cociente de los tiempos de ejecución en ambos tipos de core) y recolectar los valores de las métricas de rendimiento en M usando los contadores hardware del procesador. Debido a que los contadores son un recurso limitado, este paso puede implicar ejecutar las aplicaciones más de una vez en algunas plataformas.

3. Construir los modelos de estimación para aproximar el SF en ambos tipos de core. Para esto se debe tener en cuenta el *speedup factor medio* y los valores promedio de las métricas de rendimiento obtenidos durante la ejecución de las aplicaciones en AP sobre los dos tipos de core. A continuación, se aplica regresión aditiva [15] para aproximar el SF a partir de los valores de las métricas. Para esta tarea utilizamos el motor de regresión aditiva proporcionado por la herramienta WEKA [21]. Como resultado de este paso se obtienen dos modelos de estimación: uno para estimar el SF a partir de las métricas de los cores rápidos y otro para la estimación a partir de las métricas de cores lentos.

4. En función de los modelos obtenidos en el paso anterior, identificar el subconjunto de métricas ($SM \subseteq M$) con mayores coeficientes de correlación que puedan ser monitorizadas simultáneamente en la plataforma en cuestión. En particular, el número de métricas en SM no debería requerir más contadores hardware que los disponibles en la plataforma, de lo contrario será necesario realizar multiplexación de eventos cuando se estima el SF (es decir, implicaría monitorizar conjuntos diferentes de métricas siguiendo una estrategia *round-robin*).

5. Por último, construir los modelos de estimación de SF para ambos tipos de core realizando las mismas acciones que en el *paso 3*. Sin embargo, ahora sólo deben utilizarse los valores obtenidos para las métricas en SM. Como resultado se obtienen dos modelos de estimación (uno para cada tipo de core) que sólo dependen de las métricas en SM.

De aquí en adelante nos referiremos a esta metodología como estrategia *Overall-SF*.



(a) Core rápido

(b) Core lento

Figura 6.5: Predicción de SF sobre los cores rápidos y lentos en el sistema AMD con la estrategia Overall-SF. Los puntos ubicados en la diagonal representan estimaciones perfectas.

Para construir los modelos de estimación de *SF* en nuestras plataformas experimentales empleamos algunas aplicaciones de la suite de *benchmarks* SPEC CPU2000 y CPU2006, que constituyen nuestro conjunto *AP*. Durante la ejecución de estas aplicaciones se recolectó el valor de una amplia gama de métricas de rendimiento, tales como el número de instrucciones por ciclo (IPC), fallos y accesos a la LLC (último nivel de cache) por 1K instrucciones, la tasa de fallos de predicción de saltos, el número de fallos de DTLB y ITLB cada 1M instrucciones, etc.

La figura 6.5 muestra los valores de *speedup factor* observados y estimados tanto en el core rápido como en el core lento del sistema AMD-12. En esta plataforma, la estrategia *Overall-SF* genera modelos de estimación precisos y sencillos que se basan en métricas relacionadas con el acceso a la cache y a memoria principal. La simplicidad de este modelo se debe a que los cores difieren sólo en la frecuencia del procesador. Investigaciones previas [28, 58] han puesto de de manifiesto que en este escenario las tasas de fallos y de accesos a la LLC exhiben una correlación negativa con el *SF*. En particular, estos modelos alcanzan coeficientes de correlación de $0,97$ y $0,96$ al predecir el *SF* en el core rápido y en el core lento, respectivamente.

Lamentablemente, la estrategia *Overall-SF* no permite obtener un modelo preciso de estimación de *SF* en el prototipo de Intel QuickIA. Esto se debe principalmente a las profundas diferencias entre los cores (microarquitectura, tamaño de caché, frecuencia del procesador, etc.) que esta arquitectura posee, y al escaso número de contadores hardware disponibles. Concretamente, en este sistema sólo se pueden monitorizar tres métricas de rendimiento de forma simultánea en cualquier tipo de core. Con esta restricción, los modelos generados con *Overall-SF* en ambos tipos de cores exhiben coeficientes de correlación por debajo de $0,80$. En esta plataforma también observamos que ninguna de las métricas de rendimiento obtenidas durante la ejecución de las aplicaciones exhibe una clara correlación con el *speedup factor* medio (ver figura 6.6). Por esta razón la técnica de regresión aditiva empleada no permite

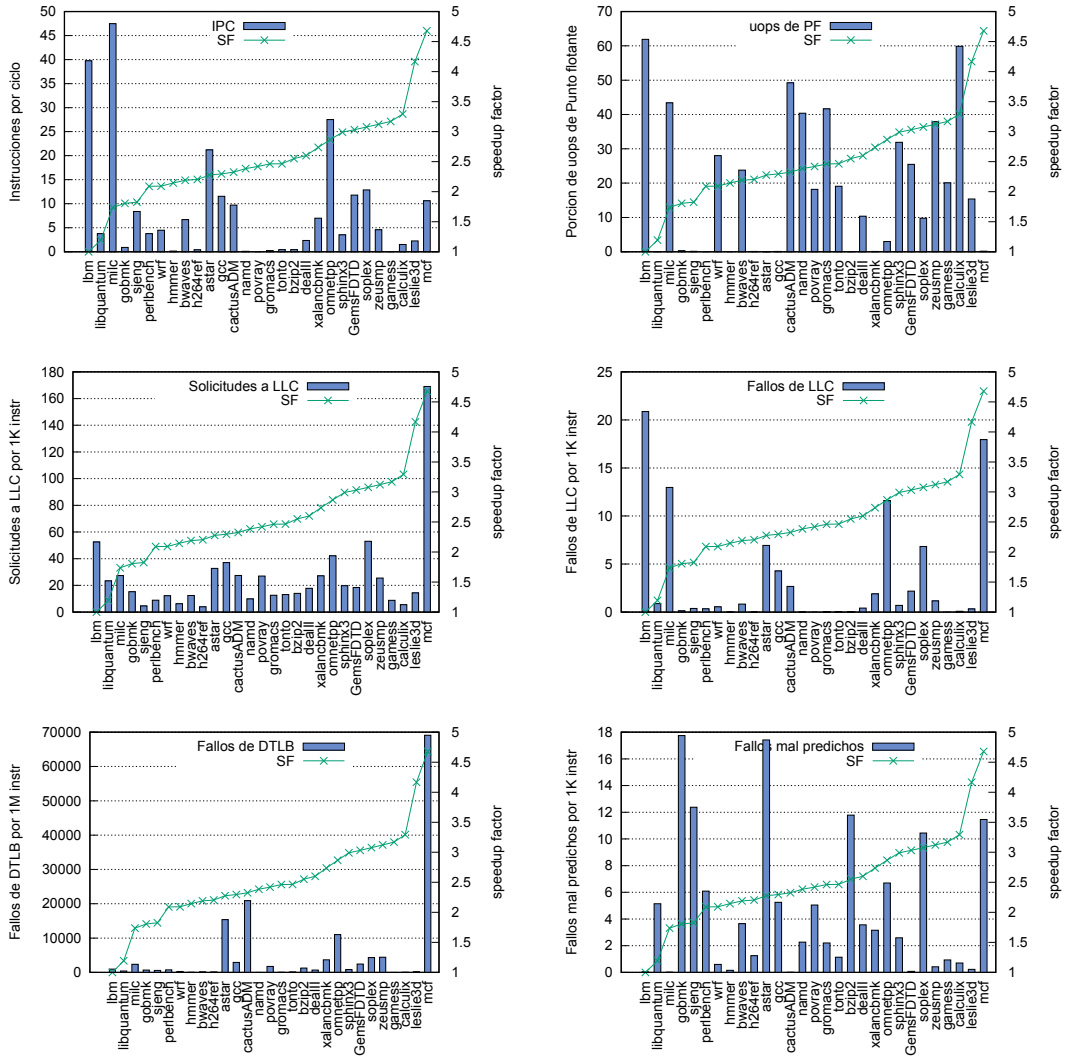
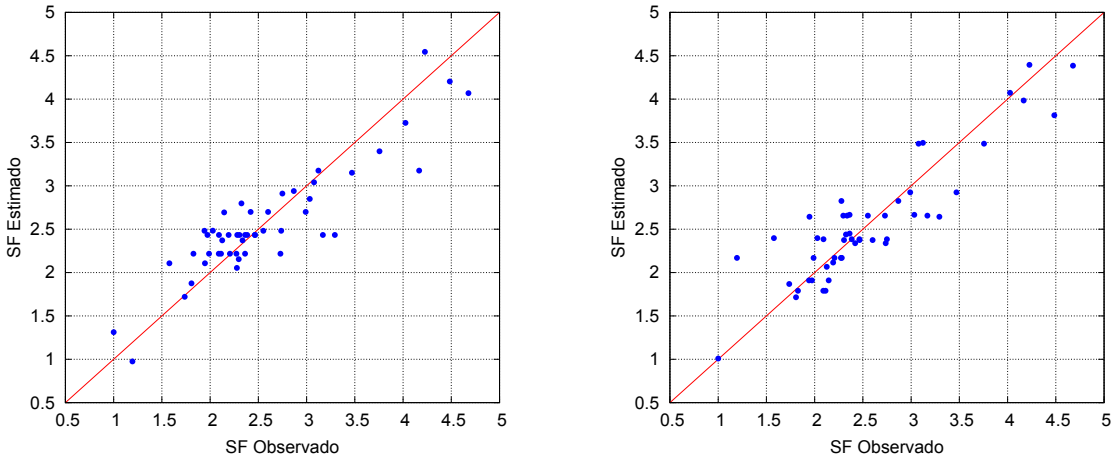


Figura 6.6: Speedup factor vs. distintas métricas de rendimiento en el core rápido para la suite de benchmarks SPEC CPU2006 ejecutadas sobre el prototipo de Intel QuickIA.

construir modelos de estimación precisos.

También analizamos la efectividad de los modelos de estimación de *SF* que dependen de más de tres métricas de rendimiento en el QuickIA de Intel. Nótese que para usar estos modelos de rendimiento en tiempo de ejecución, el planificador debe realizar multiplexación de eventos, que permite eliminar la restricción del número de métricas que pueden monitorizarse en el SO. Al disponer de mayor número de métricas es posible obtener modelos de estimación con un coeficiente de correlación de hasta 0,90 en ambos tipos de core. La figura 6.7 muestra los valores de *SF* observados y estimados en ambos tipos de core utilizando estos modelos. Además de los relativamente bajos coeficientes de correlación, detectamos que estos modelos no proporcionan estimaciones precisas para las distintas fases de ejecución de los programas. Más concretamente, observamos que algunas aplicaciones que se ejecutan



(a) Core rápido

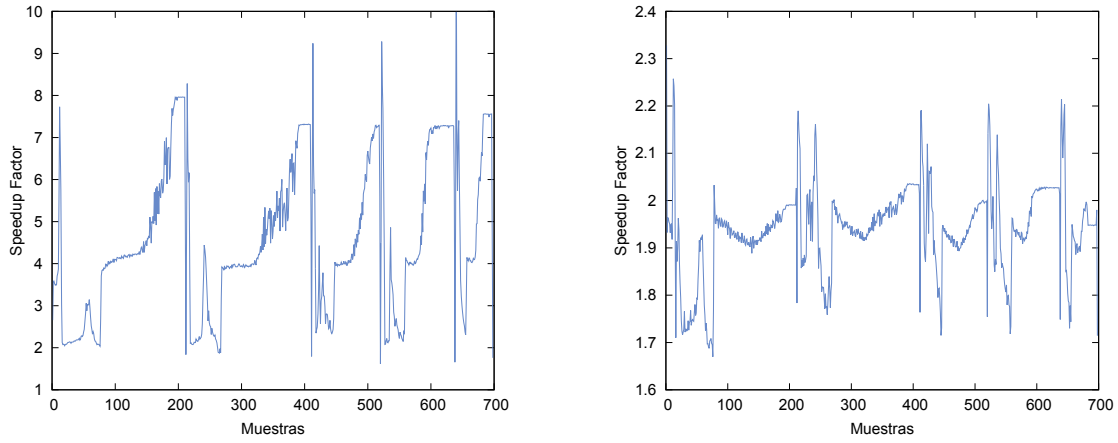
(b) Core lento

Figura 6.7: Predicción de SF sobre los cores rápido y lento en el sistema QuickIA con la estrategia Overall-SF para aplicaciones de la suite de benchmarks SPEC CPU.

sobre QuickIA exhiben fases de grano grueso con valores de *SF* muy superiores al máximo *SF* promedio observado para los *benchmarks* de la suite SPEC CPU (4,7 en esta plataforma). Por ejemplo, la figura 6.8a muestra el *SF* a lo largo del tiempo para el *benchmark mcf* de la suite SPEC CPU2006. Esta aplicación presenta fases de larga duración con un valor de *SF* de hasta 8. Para aplicaciones como ésta, los modelos obtenidos con la estrategia *Overall-SF* proporcionan estimaciones muy imprecisas para fases de ejecución con un *SF* tan elevado.

Los modelos derivados para el sistema de AMD-12 no poseen esta limitación, y ofrecen estimaciones precisas para diferentes fases de ejecución en las aplicaciones. Nótese que los *benchmarks* en la plataforma AMD no presentan fases de grano grueso con un valor de *SF* superior al el máximo *SF* promedio observado en esta plataforma (2,625). Como se muestra en la figura 6.8b, el *benchmark mcf* no atraviesa fases de larga duración con $SF > 2{,}625$ en dicho sistema.

Para generar modelos precisos de estimación de *SF* sobre el prototipo QuickIA, ideamos una variante de la estrategia *Overall-SF*. En lugar de emplear el *speedup factor* promedio y los valores de las métricas de rendimiento de la ejecución completa de los benchmarks para generar los modelos, la nueva estrategia utiliza información asociada a las diferentes fases de programa. Nos referiremos a esta nueva estrategia como *Phase-SF*. En particular, *Phase-SF* emplea el valor del *speedup factor* y de las métricas de rendimiento por cada fase de ejecución individual que atraviesa una aplicación a lo largo del tiempo. Para recolectar esta información, tomamos muestras con los contadores hardware por cada 200 millones de instrucciones en ambos tipos de core para cada benchmark. Este intervalo de muestreo nos permite capturar de forma efectiva las fases de programa de grano grueso y filtrar los picos de *SF* (fases de muy corta duración). Nótese que el *SF* para una cierta ventana de instrucciones de una aplicación es el ratio de los valores de IPS recolectados en el core rápido y en el core lento para esa



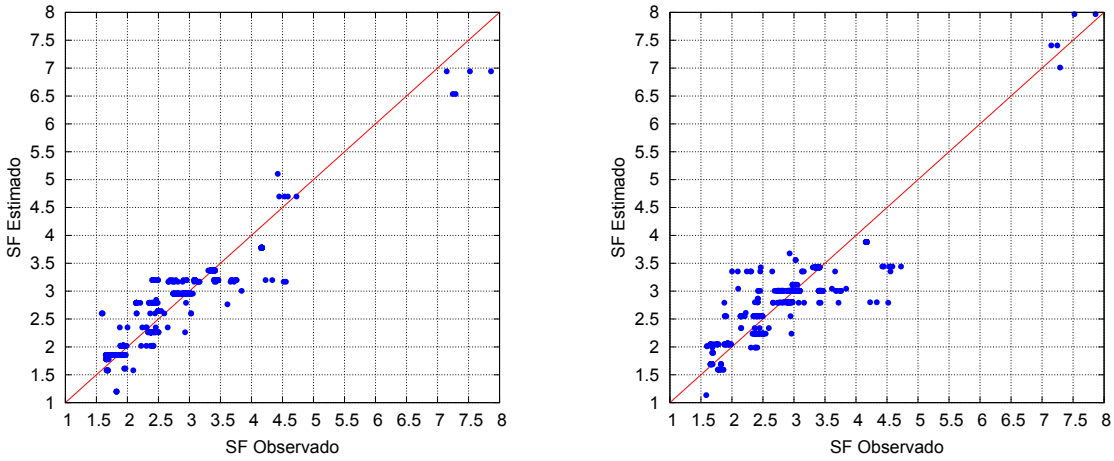
(a) QuickIA (b) AMD

Figura 6.8: SF observados a lo largo del tiempo para el benchmark mcf sobre los sistemas Intel QuickIA y AMD.

ventana de instrucciones.

A partir de las muestras de *SF* para las distintas ventanas de instrucciones de una aplicación, procedemos a identificar las fases de *SF* de grano grueso del programa. El mecanismo para dividir las muestras de *SF* en fases se describe en el anexo B. Una vez detectadas las fases de *SF* de grano grueso de una aplicación generamos un resumen compacto de cada fase. El resumen es esencialmente una tupla formada por la media geométrica de los valores de cada métrica de rendimiento durante la fase de ejecución y la media geométrica de los valores de *SF* registrados en esta fase. Por último, utilizamos los resúmenes de fases de *SF* obtenidos para cada aplicación como entrada al motor de regresión aditiva de WEKA. Esto permite generar los modelos finales de estimación de *SF* para ambos tipos core.

La figura 6.9 muestra la predicción de *SF* sobre el prototipo QuickIA de Intel utilizando los modelos de estimación obtenidos con la estrategia *Phase-SF*. Con estos modelos, los coeficientes de correlación obtenidos para la estimación sobre el core rápido y el core lento son 0,95 y 0,94, respectivamente. Para generar los modelos en esta plataforma, empleamos información de rendimiento de 742 fases de *SF* extraídas de los *benchmarks* de SPEC CPU. No obstante, los datos obtenidos revelan que para generar un modelo de estimación para el core rápido con una precisión similar al obtenido con 742 fases, basta con utilizar información de 500 fases diversas de *SF*. Por el contrario, para obtener un modelo de estimación para el core lento con un coeficiente de correlación no inferior a 0,94 es necesario recolectar información de al menos 700 fases de *SF*. El motor de predicción de regresión aditiva que utilizamos permite descartar algunas métricas de rendimiento de los modelos de estimación finales. Aún así los modelos dependen de más de tres métricas de rendimiento en ambos tipos de core (mostradas en la tabla 6.1). Por esta razón la implementación del planificador necesita en este caso realizar multiplexación de eventos para estimar el *SF online*. En la siguiente sección



(a) Core rápido

(b) Core lento

Figura 6.9: Predicción de SF basada en fases sobre los cores rápido y lento en el sistema QuickIA para aplicaciones de la suite de benchmarks SPEC CPU. El modelo asociado ha sido obtenido utilizando la estrategia Phase-SF.

Métricas de rendimiento	Modelo de core rápido	Modelo de core lento
Instrucciones retiradas por ciclo (IPC)	✓	✓
Accesos a la L2 por cada 1K instr.	✓	✓
Fallos de L2 por cada 1K instr.	✓	✓
Instrucciones de salto por cada 1K instr.	✓	✓
Fallos de predicción de saltos por cada 1K instr.	✓	✓
Fallos de ITLB por cada 1M instr.		✓
Fallos de DTLB por cada 1M instr.		✓

Tabla 6.1: Métricas de rendimiento utilizadas para predecir el SF el sistema Intel QuickIA

proporcionamos más detalles sobre este aspecto.

Para concluir, vale la pena mencionar que el análisis *offline* requerido para construir modelos de estimación de *SF* debe realizarse una única vez en cada plataforma asimétrica. Para garantizar la aplicación de nuestra metodología de manera transparente en un sistema de producción, los experimentos asociados podrían lanzarse automáticamente por el SO cuando éste arranca por primera vez en la plataforma. Alternativamente, los modelos de estimación podrían regenerarse de manera periódica y automática, aprovechando períodos de inactividad del sistema para recolectar muestras de rendimiento de aplicaciones adicionales. En nuestros experimentos, realizamos una ejecución completa de un subconjunto de *benchmarks* SPEC CPU sobre diferentes tipos de cores del sistema, con el objetivo de realizar una caracterización completa del rendimiento de estos *benchmarks* en cada configuración AMP evaluada. Esto puede demandar mucho tiempo, especialmente en los cores lentos del QuickIA (Intel Atom), por lo que no constituye una buena estrategia en la práctica. Sin embargo, nuestros resultados revelan que para construir modelos de estimación con precisión similar a los obtenidos, no es necesario realizar una ejecución completa de los *benchmarks*. En la práctica, la mayor parte de

las fases representativas de programas provienen de unos pocos *benchmarks* de corta duración y con múltiples fases, tales como *soplex*, *astar* o *gcc*. Algunas otras fases representativas aparecen al comienzo de la ejecución de *benchmarks* con un tiempo de ejecución prolongado, tales como *lbm* o *libquantum*, que exhiben un comportamiento extremadamente regular. Por lo tanto, para obtener modelos de estimación con precisión similar, basta recolectar información para sólo un número reducido de ventanas de instrucciones de un conjunto representativo de *benckmarks*.

6.3.3. Implementación: utilizando los modelos de estimación de speedup factor en tiempo de ejecución

Para implementar los modelos de estimación derivados para las plataformas Intel QuickIA y AMD-12, creamos dos *módulos de monitorización* independientes usando la herramienta PMCTrack [55]. Los módulos de monitorización (implementados en un módulo cargable del kernel) alimentan al planificador con estimaciones de *SF* por hilo en tiempo de ejecución. Utilizando esta estrategia, la implementación del algoritmo de planificación en sí (creada dentro del kernel del SO) se mantiene independiente de la arquitectura y completamente desacoplada de la técnica subyacente para obtener los SFs *online* [55]. Para obtener estimaciones de *SF*, el módulo de monitorización en cuestión monitoriza continuamente las métricas de rendimiento necesarias utilizando los contadores hardware del procesador, y aplica el modelo de estimación. En nuestra plataforma experimental, tomamos muestras de los contadores cada $200ms$ por hilo; este intervalo de muestreo fue utilizado en trabajos previos de planificación sobre AMPs [47][59]. Para este valor observamos que el *overhead* asociado al muestreo y a la estimación de SFs es despreciable.

A diferencia de la estimación de *SF* en la plataforma AMD, determinar los SFs en el Intel QuickIA requiere monitorizar un conjunto de métricas de rendimiento cuyos valores no pueden recolectarse simultáneamente usando el número de contadores de rendimiento disponibles. Por esto, recabar la información necesaria para el modelo de estimación sobre el QuickIA implica monitorizar diferentes conjuntos de eventos hardware siguiendo una estrategia *round-robin* (es decir, multiplexación de eventos). En esta plataforma, es necesario completar dos intervalos de muestreo de los contadores para monitorizar todas las métricas en los cores rápidos, y se necesitan tres intervalos de muestreo para hacerlo en los cores lentos. Una vez que todas las métricas de rendimiento requeridas han sido recolectadas en un tipo de core especifico utilizando los intervalos de muestreo necesarios, los valores de las métricas se emplean para predecir el *SF* del hilo. Este proceso se repite continuamente durante la ejecución del hilo para alimentar al planificador con estimaciones de *SF* de las distintas fases de ejecución.

Al implementar esta estrategia en el QuickIA detectamos un problema significativo. Las métricas de rendimiento monitorizadas en intervalos de muestreo consecutivos en un core dado pueden no pertenecer a la misma fase de ejecución del programa. Lamentablemente,

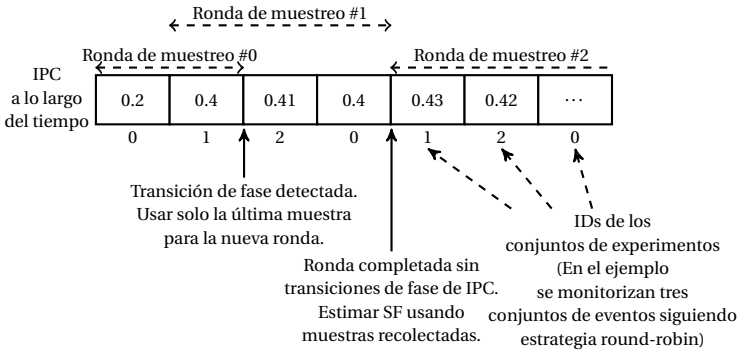


Figura 6.10: Detección de transiciones de fase al utilizar multiplexación de eventos

utilizar valores de métricas de eventos pertenecientes a diferentes fases de ejecución lleva a estimaciones de *SF* muy imprecisas. Para solucionar este problema, ampliamos la estrategia de predicción básica con una técnica que permite detectar transiciones entre distintas fases de ejecución. Esta técnica es similar a la propuesta presentada en [8], que se evaluó con un simulador. A grandes rasgos, la técnica de detección de fases funciona de la siguiente forma. Para cada hilo, mantenemos la *media móvil* del IPC. Para esto, siempre monitorizamos el número de instrucciones por ciclo junto con otras métricas en un conjunto de eventos particular. Si se detecta una variación abrupta del IPC en la última muestra (con respecto a la *media móvil* del IPC), asumimos que la última muestra pertenece a una nueva fase de ejecución. Si este es el caso, las muestras recolectadas previamente dentro de la misma ronda de muestreo se descartan a la hora de estimar el *SF*, y comenzamos una nueva ronda de muestreo. Por el contrario, al completar una ronda de muestreo sin detectar transiciones de fase, entonces las muestras recolectadas se utilizan para obtener una nueva predicción del *SF* aplicando el modelo de estimación. Para facilitar la comprensión de esta técnica, la figura 6.10 muestra como funciona el mecanismo de detección de transiciones de fase en un escenario hipotético donde se necesitan tres intervalos de muestreo consecutivos sin cambios de fase (tres conjuntos de eventos) para poder recolectar las muestras necesarias para la aplicación del modelo de estimación. En nuestras plataformas experimentales observamos que esta técnica resulta muy efectiva, y nos permite obtener estimaciones de *SF* más precisas a lo largo del tiempo.

6.4. Evaluación experimental

Para evaluar la eficacia de ACFS, realizamos una comparación con otros algoritmos para AMPs, como nuestra propuesta previa Prop-SP, descrita en el capítulo 4, y otros algoritmos conscientes de la asimetría, como HSP [28, 54], la versión de EQual-Progress (EQP) [11] que emplea *IPC-sampling*, RR [8] y A-DWRR [33]. Todas estas propuestas previas se describen en el capítulo 3. Para llevar a cabo la evaluación experimental, implementamos todos los algoritmos en el kernel Linux, sobre la clase de planificación AMP.

Al igual que ACFS, los planificadores HSP, Prop-SP y EQP toman decisiones teniendo en cuenta el *SF* de los hilos, que se aproxima utilizando los contadores hardware del procesador. Para determinar el *SFs* de un hilo en tiempo de ejecución sobre hardware multicore asimétrico real, EQP [11] utiliza *IPC-sampling*, medición directa del *SF*, (ver sección 3.2.1.1). A pesar de que HSP y Prop-SP podrían usar teóricamente cualquier mecanismo para aproximar el *SF*, en nuestros experimentos optamos por utilizar estimación de *SF* en lugar de *IPC-sampling* para estos planificadores, ya que observamos que los modelos de estimación permiten a estas estrategias lograr mayor rendimiento global que el obtenido al usar medición directa del *SF*.

Para realizar nuestra evaluación experimental utilizamos cargas de trabajo multiprogramadas compuestas por aplicaciones de diversas suites de benchmarks para HPC (SPEC CPU2006, OMP 2001, PARSEC y Minebench). Experimentamos también con *BLAST*, un benchmark de bioinformática, y *FFTW3D*, un programa que realiza la transformada rápida de Fourier. En todos los experimentos el número total de hilos de la carga de trabajo coincide con el número de cores de la plataforma, como en trabajos previos en AMPs que también emplean cargas de trabajo intensivas en cómputo[28, 59, 46]. Asimismo, en los experimentos garantizamos que todas las aplicaciones se inician simultáneamente y, cuando una aplicación termina, se reinicia varias veces hasta que la aplicación de mayor duración del conjunto se completa en tres ocasiones. Finalizada la ejecución de la carga de trabajo, calculamos el *ASP* y *unfairness* para el algoritmo de planificación en cuestión, utilizando la media geométrica de los tiempos de ejecución para cada aplicación.

El resto de esta sección se divide en cuatro partes. En la sección 6.4.1 analizamos la efectividad de los distintos algoritmos de planificación para cargas de trabajo donde todas las aplicaciones tienen la misma prioridad. La sección 6.4.2 cubre escenarios donde las aplicaciones tienen asignadas prioridades diferentes. En la sección 6.4.3 mostramos cómo se comporta ACFS al ajustar el parámetro de configuración *unfairness_factor*. Finalmente, en la sección 6.5 realizamos un estudio de sensibilidad que muestra el impacto de la frecuencia de intercambio de hilos en el rendimiento global y la justicia bajo ACFS.

6.4.1. Aplicaciones con la misma prioridad

Para crear las cargas de trabajo, clasificamos las aplicaciones en las 9 subcategorías descritas en la sección 4.4.1, y creadas en base al SF de los hilos (categorías H, M y L) y teniendo en cuenta su grado de paralelismo a nivel de hilo (categorías HP, PS o ST).

6.4.1.1. Cargas de trabajo compuestas por aplicaciones secuenciales

Comenzamos nuestro análisis con el estudio de los resultados obtenidos para cargas de trabajo formadas por múltiples aplicaciones secuenciales, que se ejecutan sobre la configuración 2F-2S (Intel QuickIA). La tabla 6.2 muestra las cargas de trabajo seleccionadas, que representan mezclas de programas con diferentes rangos de *SF*. El nombre de cada carga de

Tabla 6.2: Cargas de trabajo multi-aplicación compuestas por programas secuenciales

Carga	Benchmarks
4STH	calculix, gamess, GemsFDTD, bzip2
3STH-1STM	calculix, GemsFDTD, bzip2, h264ref
3STH-1STL	gamess, GemsFDTD, bzip2, sjeng
3STH-1STL$_B$	calculix, gamess, sphinx3, sjeng
2STH-2STM	gamess, soplex, povray, h264ref
2STH-2STL	mcf, calculix, sjeng, gobmk
2STH-2STL$_B$	gamess, sphinx3, gobmk, libquantum
1STH-1STM-2STL	mcf, h264ref, sjeng, gobmk
2STM-2STL	namd, h264ref, gobmk, libquantum

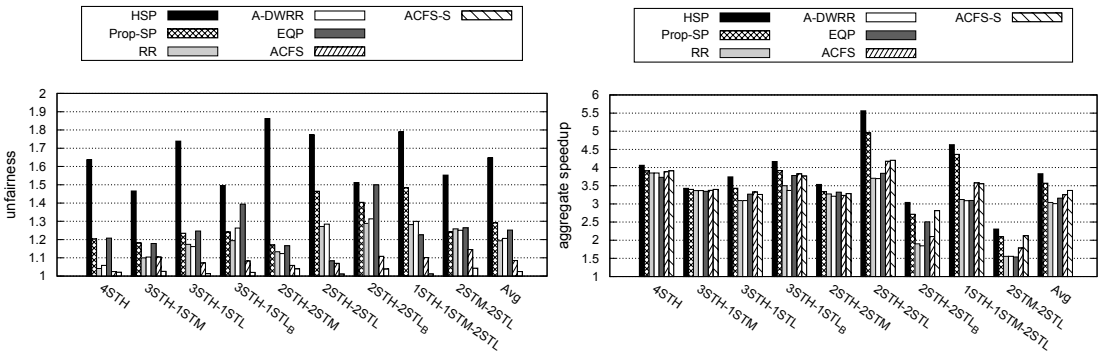


Figura 6.11: ASP y Unfairness sobre 2F-2S (Intel QuickIA).

trabajo que se muestra en la tabla denota la categoría de cada aplicación de acuerdo al orden en que aparecen en la fila. Por ejemplo, en la carga 2STH-2STL, mcf y calculix son aplicaciones STH (*Single-Threaded High-SF*) y sjeng06 y gobmk son aplicaciones STL (*Single-Threaded Low-SF*). Nótese que las categorías de *SF* (H, M y L) son dependientes de la arquitectura. Por ejemplo, el benchmark gobmk se clasifica como un programa *Low-SF* en 2F-2S (Intel QuickIA) y como un programa *High-SF* en 2F-10S (plataforma AMD-12). Para distinguir entre cargas de trabajo compuestas por exactamente las mismas categorías de referencia, se añadió un sufijo al nombre. Por ejemplo: existen dos cargas de trabajo en la categoría 3STH-1STL, la primera no lleva sufijo y la segunda el sufijo _B (3STH-1STL_B).

La figura 6.11 muestra los resultados obtenidos para las cargas de trabajo de la tabla 6.2. Como se observa en la figura 6.11, el algoritmo de planificación que optimiza el rendimiento (HSP), consigue en todos los casos los mejores valores de *ASP*, pero a expensas de obtener los peores valores de *unfairness*. Los algoritmos RR y A-DWRR funcionan de manera similar en la mayoría de los casos. Ambos algoritmos distribuyen el tiempo de uso de core rápido proporcionalmente entre los hilos. Sin embargo, ésta distribución puede conducir a una degradación en la justicia y en el rendimiento global, especialmente para cargas de trabajo con un amplio rango de *speedups* entre aplicaciones (por ejemplo: 3STH-1STL, 2STH-2STL). Los resultados también muestran que el algoritmo Prop-SP ofrece un mejor compromiso rendimiento-justicia que HSP. Sin embargo, en algunos casos se comporta de forma injusta (valor alto de *unfairness*), como por ejemplo en 2STH-2STL.

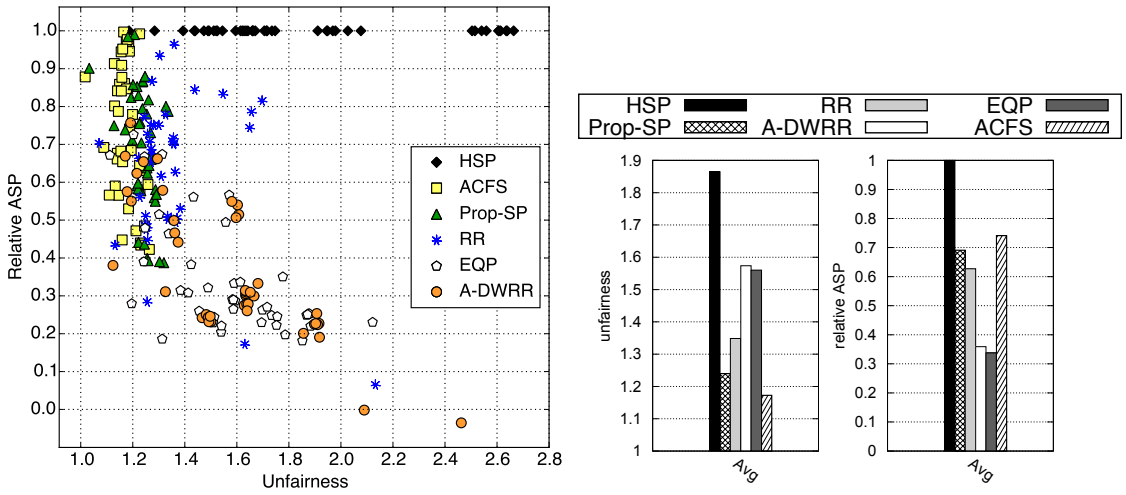
Ahora nos centramos en la discusión de los resultados para EQP y ACFS, que intentan optimizar la justicia. Claramente EQP no es capaz de obtener valores de *unfairness* menores que RR o A-DWRR para todas las cargas de trabajo, por lo tanto no logra su principal objetivo. Observamos que esto se debe a las imprecisiones asociadas con el mecanismo empleado por EQP para aproximar la degradación acumulada (*slowdown*) por los hilos de ejecución. Por otra parte, EQP está sujeto a fallos de predicción de *SF* ligados al uso de *IPC-sampling*. Se ha demostrado que esta estrategia para aproximar el *SF* produce con frecuencia valores de *SF* imprecisos, debido al hecho de que los valores de *IPC* medidos en cada tipo de core (y usados para aproximar el SF) pueden pertenecer a distintas fases de ejecución del programa[60]. Hemos observado que estos valores imprecisos se manifiestan sobre todo en las últimas cuatro cargas de trabajo. Por el contrario, nuestro algoritmo de planificación (ACFS) no está sujeto a estos problemas, ya que predice el *SF* de un hilo por medio de un modelo de estimación que utiliza métricas de rendimiento obtenidas únicamente en el tipo de core actual (sin requerir migraciones al core opuesto para este propósito). Los resultados revelan que, a pesar de las imprecisiones que podrían existir en el modelo de estimación de *SF*, ACFS obtiene para todas las cargas de trabajo exploradas los mejores valores de *unfairness*. En promedio nuestra propuesta reduce el *unfairness* en un 10 % en comparación con RR y A-DWRR, y en un 13 % en comparación con EQP, y al mismo tiempo garantiza un mayor rendimiento global que estos algoritmos.

Finalmente, para evaluar el impacto que tienen los fallos de estimación de *SF* en ACFS, experimentamos con una versión estática de ACFS (ACFS-S). En esta versión, el planificador es alimentado con valores de *SF* de la aplicación obtenidos *offline* para su ejecución completa. Como puede observarse, usar estimaciones perfectas de *SF* permiten reducir significativamente los valores de *unfairness*. Este hecho pone de manifiesto la importancia de obtener una estimación de *SF* precisa para garantizar justicia en un AMP.

6.4.1.2. Cargas de trabajo compuestas por aplicaciones secuenciales y multi-hilo

Continuamos nuestro análisis evaluando la efectividad de los distintos algoritmos de planificación al utilizar cargas de trabajo compuestas por aplicaciones paralelas y secuenciales. Para experimentar con este tipo de cargas utilizamos la configuración 2F-10S sobre la plataforma AMD-12, debido a que el prototipo QuickIA de Intel no es apropiado por el escaso número de cores que posee.

En este nuevo escenario, existe un conjunto más amplio de tipos de cargas de trabajo a explorar, debido a la existencia de hasta 9 clases distintas de aplicación: 3 clases de paralelismo (ST, HP y PS) con 3 subclases de *SF* cada una (H, M y L). Para atender a esta diversidad, generamos 44 combinaciones de programas, combinando 14 aplicaciones de diferentes categorías. Al generar las cargas de trabajo, nos aseguramos de incluir al menos una aplicación multi-hilo en cada carga. En general, observamos que estas cargas de trabajo exhiben un amplio rango de *speedups fast-to-slow* entre aplicaciones, en comparación a aquellas cargas de trabajo que están compuestas sólo por aplicaciones secuenciales. En particular, muchas cargas combinan



(a) (b)

Figura 6.12: Resultados para cargas de trabajo compuestas por aplicaciones secuenciales y multihilo sobre la configuración 2F-10S (AMD).

aplicaciones paralelas que experimentan un *speedup* muy bajo al utilizar los escasos cores rápidos (p.ej., aplicaciones HP) con aplicaciones que experimentan ganancias de rendimiento significativas al utilizar este tipo de cores (por ejemplo, aplicaciones secuenciales).

La figura 6.12a muestra la relación entre el *unfairness* y el rendimiento global para todas las cargas de trabajo consideradas. Para mayor claridad, todos los valores de rendimiento global mostrados (*ASP*) están normalizados con respecto al rendimiento global obtenido por el planificador HSP. Nótese que cuanto más cerca se encuentra un determinado punto (carga de trabajo) a la esquina superior izquierda, mejor es el compromiso rendimiento-justicia. Para ilustrar el comportamiento general de los distintos algoritmos, la figura 6.12b muestra los valores promedio de *unfairness* y ASP relativo para todas las cargas de trabajo. Además, la figura 6.13 muestra los valores individuales de *unfairness* y rendimiento global para 10 cargas de trabajo seleccionadas de distintas categorías, que representan puntos muy diferentes en la figura 6.12a. La tabla 6.3 muestra la composición de estas cargas de trabajo. Nótese que para aplicaciones multi-hilo, el número entre paréntesis que se muestra junto al nombre de programa de la tabla indica el número de hilos con el que se ejecuta.

Si analizamos los resultados de HSP y ACFS de las figuras 6.12 y 6.13 sobre la configuración 2F-10S, podemos observar una tendencia similar a los resultados obtenidos sobre la configuración 2F-2S: ACFS obtiene el mejor valor de *unfairness* para la mayoría de las cargas de trabajo, mientras que HSP obtiene valores de *ASP* ligeramente mayores que ACFS a expensas de degradar significativamente la justicia. En este escenario observamos que, en promedio, ACFS obtiene mejoras más modestas en cuanto a justicia que Prop-SP (5,5 %). Esta mejora es mucho más notoria en cargas de trabajo que incluyen aplicaciones secuenciales (16 %). Para la mayoría de las cargas de trabajo de la tabla 6.3, detectamos que Prop-SP realiza una distribución de ciclos de core rápido muy similar a ACFS. Por esta razón la justicia y el rendi-

Tabla 6.3: Subconjunto de cargas de trabajo compuestas por aplicaciones secuenciales y multi-hilo

Carga de trabajo	Benchmarks
2STH-1HPH-1HPM	gamess, hmmer, fma3d_m(5), wupwise_m (5)
3STH-1HPH	hmmer, gobmk, h264ref, fma3d_m(9)
2STH-1STL-1HPH	perlbench, soplex, h264ref, fma3d_m(9)
3ST{H,M,L}-1PSH	gamess, astar, soplex, blackscholes(9)
2STH-1PSH-1HPM	hmmer, perlbench, semphy(5), wupwise_m(5)
1STH-1PSH-1PSL	gobmk, BLAST(6), FFTW3D(5)
2PSH-1PSL	BLAST(4), semphy(4), FFTW3D(4)
1PSH-1PSL	semphy(6), FFTW3D(6)
2PSH-1HPM	BLAST(4), semphy(4), wupwise_m(4)
1PSH-1HPH	BLAST(6), fma3d_m(6)

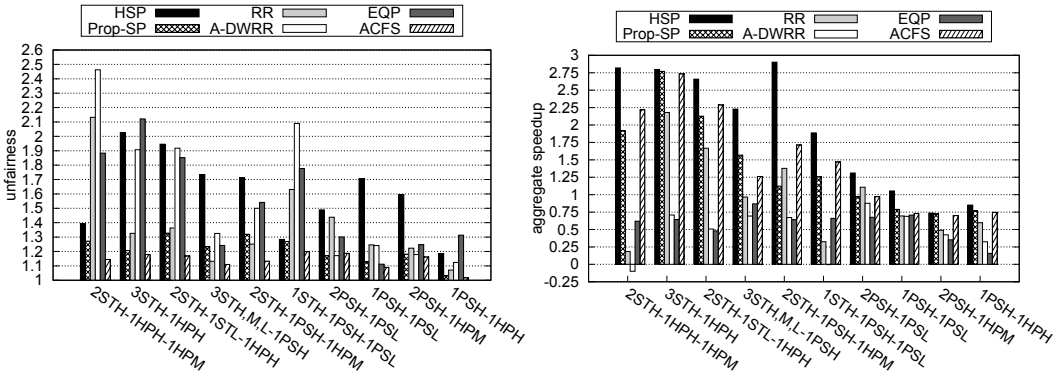


Figura 6.13: Justicia y rendimiento global para las cargas de trabajo seleccionada sobre 2B-10S (plataforma AMD)

miento son similares. En particular, si se asignan ciclos de core rápido de forma proporcional al *speedup* de la aplicación (Prop-SP), es posible alcanzar mejores valores de justicia para cargas de trabajo que incluyen tanto aplicaciones secuenciales como multi-hilo.

Mientras que los resultados asociados a los planificadores ACFS, Prop-SP y HSP exhiben una tendencia clara y consistente para todas las cargas de trabajo, observamos divergencias importantes en los resultados de EQP y A-DWRR. Los valores en la figura 6.13 explican la causa de las principales divergencias. Los resultados revelan diferencias significativas entre EQP, A-DWRR y otros planificadores orientados a justicia, especialmente para cargas de trabajo que incluyen tanto aplicaciones secuenciales y multi-hilo (las primeras 6 cargas de trabajo de la tabla 6.3). Más específicamente, EQP y A-DWRR resultan inadecuados en términos de justicia y rendimiento global para las cargas de trabajo que combinan programas secuenciales con aplicaciones altamente paralelas (tales como 3STH-1HPH o 2STH-1HPH-1HPM). De hecho, los puntos (cargas de trabajo) concentrados en la parte central inferior de la figura 6.12a para estos dos planificadores se corresponden con las cargas de trabajo que presentan dicha composición. En este contexto los pobres valores de justicia y rendimiento global obtenidos surgen porque EQP y A-DWRR toman decisiones de planificación sin tener en cuenta el número de hilos en ejecución en las distintas aplicaciones (una aproximación del grado de paralelismo a nivel de hilo). En particular, A-DWRR asegura que cada hilo de la carga de trabajo recibe el mismo tiempo de CPU escalado, independientemente de la aplicación a la cual pertenece. EQP intenta garantizar la misma degradación relativa (*slowdown*) entre los hilos considerando sólo el *SF* de los hilos individualmente, pero ignorando el número

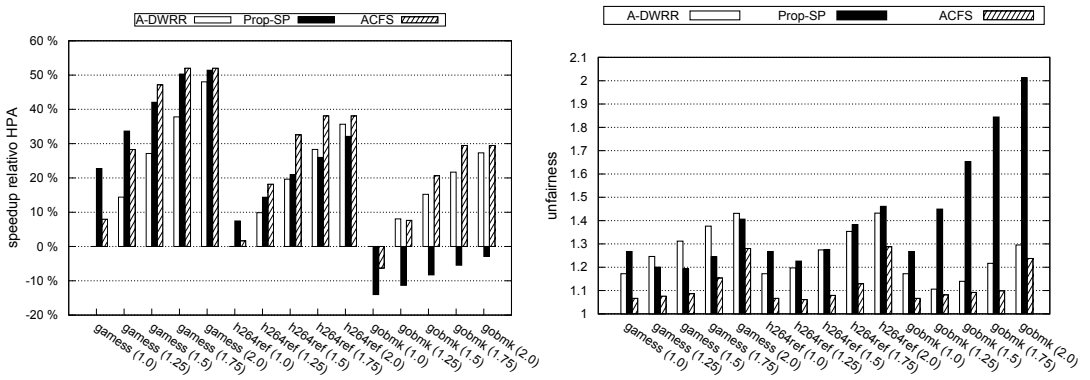
Tabla 6.4: Reducción promedio de unfairness e incremento en el rendimiento global alcanzado por ACFS con respecto a otras estrategias para el conjunto de cargas de trabajo exploradas.

ACFS vs. otros	Reducción en Unfairness	Incremento en ASP
HSP	36.65 %	-24.16 %
Prop-SP	7.25 %	2.78 %
RR	12.94 %	15.80 %
A-DWRR	23.39 %	76.10 %
EQP	23.21 %	73.65 %

de hilos en las aplicaciones asociadas. Bajo estos dos planificadores, mientras más hilos tiene la aplicación, mayor es la cantidad de ciclos de core rápido que recibe. Debido a que las aplicaciones altamente paralelas alcanzan un *speedup* bajo al utilizar los escasos cores rápidos, comparadas a las aplicaciones con paralelismo a nivel de hilo limitado [59][3], favorecer a las aplicaciones altamente paralelas sobre las aplicaciones secuenciales lleva a A-DWRR y EQP a empeorar el *unfairness* y rendimiento global, comparado a las otras estrategias.

Estos resultados nos permiten sacar la siguiente conclusión: intentar garantizar justicia entre hilos individuales en el sistema sin considerar el grado de paralelismo a nivel de hilo de la aplicación (tal como A-DWRR y EQP) no asegura el mismo progreso (misma degradación relativa) entre aplicaciones cuando la carga de trabajo incluye programas multi-hilo. Prop-SP y ACFS, por el contrario, consideran el número de hilos en la aplicación al tomar decisiones de planificación, lo que les permite obtener beneficios en los valores de justicia y rendimiento global. En particular, esto hace posible que ACFS reduzca los valores de *unfairness* un 25 % en promedio con respecto a EQP y A-DWRR, y también contribuye a mejorar significativamente el rendimiento global (ASP).

Para las cargas de trabajo que no incluyen ninguna aplicación secuencial (las últimas cuatro de la tabla 6.3), EQP y A-DWRR alcanzan valores de *unfairness* y ASP cercanos a los obtenidos por ACFS y Prop-SP. Nótese que las aplicaciones multi-hilo incluidas en estas cargas de trabajo alcanzan valores de *speedup* modestos al utilizar los escasos cores rápidos. Los valores bajos de *speedup* entre aplicaciones (observados cuando un programa se ejecuta solo en el AMP) típicamente conducen a una baja degradación relativa del rendimiento (*slowdown*) para todas las aplicaciones de la carga. En este contexto, esto garantiza valores bajos de *unfairness* y ASP, para todos los planificadores orientados a justicia, incluyendo al planificador RR. Sin embargo, RR supera a EQP y A-DWRR para cargas de trabajo que combinan una aplicación HP y varias aplicaciones secuenciales. En este escenario, RR asigna mayor cantidad de ciclos de core rápido a hilos de programas secuenciales que a hilos individuales en aplicaciones altamente paralelas (HP). Como se mencionó previamente, éste no es el caso bajo EQP y A-DWRR, que lleva a estos dos planificadores a aumentar el *unfairness* y degradar el rendimiento global, comparado a RR. A pesar de esto, ACFS sigue reduciendo el *unfairness* en un 13 % (en promedio) con respecto a RR.



(a) speedup HPA

(b) Unfairness

Figura 6.14: Resultados para aplicaciones con diferentes prioridades.

6.4.1.3. Resultados generales

La tabla 6.4 muestra la reducción promedio de *unfairness* y el incremento medio en rendimiento global obtenidos por ACFS con respecto a las demás estrategias, para todas las cargas de trabajo exploradas, 9 de ellas formadas únicamente por programas secuenciales, y las 44 restantes constituidas por aplicaciones secuenciales y multi-hilo. Como puede observarse, ACFS reduce el *unfairness* significativamente, con respecto a las demás estrategias orientadas a justicia: 23 % respecto a EQP y A-DWRR, 13 % comparado a RR, y 7 % sobre Prop-SP. Además ACFS es capaz de alcanzar ganancias significativas en rendimiento global (ASP), comparado con estas estrategias.

6.4.2. Aplicaciones con distintas prioridades

En esta sección evaluamos la efectividad de algunas de las estrategias de planificación que permiten al usuario asignar distintas prioridades a las aplicaciones: A-DWRR, Prop-SP y ACFS. Para ello, experimentamos sobre una configuración 2F-2S con una carga de trabajo compuesta por dos aplicaciones con *speedup* alto (gamess y bzip2), una aplicación con *speedup* medio (*h264ref*) y una aplicación con *speedup* bajo (*gobmk*). Utilizando esta carga de trabajo, pudimos explorar cómo aceleran distintos tipos de aplicaciones a medida que aumentamos su prioridad, y analizamos cómo se ve afectada la justicia bajo los distintos algoritmos de planificación.

Nuestro experimento consistió en aumentar gradualmente la prioridad de una aplicación de alta prioridad (*HPA*) seleccionada, manteniendo la prioridad del resto de las aplicaciones al valor por defecto. Para cada aplicación *HPA*, aumentamos gradualmente su prioridad tal que el peso asociado (W_t) se incremente en pasos del 25 %.

La figura 6.14 muestra los resultados de nuestra evaluación. En el eje horizontal indicamos el nombre de la aplicación *HPA* seleccionada. Entre paréntesis especificamos el peso

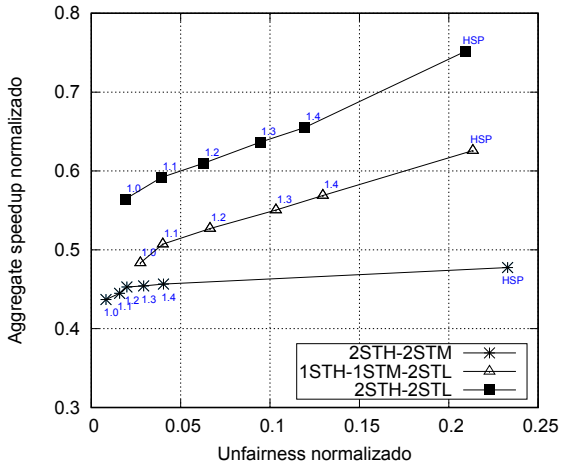


Figura 6.15: ASP y Unfairness normalizado para distintos valores de UF.

asociado W_t que se deriva directamente de la prioridad de la aplicación. Para las diferentes combinaciones de prioridades y algoritmos de planificación mostramos el *speedup relativo* de la aplicación *HPA* y *unfairness*. Al considerar las prioridades de usuario, es necesario tener en cuenta la prioridad de la aplicación en la métrica de *unfairness* [13]. Para esto, reemplazamos el *slowdown* en la expresión del *unfairness* (ecuación 3.3, sección 3.1) por su homólogo ponderado ($W_t \cdot Slowdown_{app}$). El *speedup HPA* está normalizado a A-DWRR en el escenario donde todas las aplicaciones tienen la misma prioridad. De esta manera, podemos hacer un seguimiento de cuánto acelera la aplicación *HPA* en función de la prioridad. Dado que observamos tendencias muy similares para las dos aplicaciones secuenciales con *speedup* alto, omitimos los resultados para *bzip2*.

En un escenario donde las aplicaciones tienen la misma prioridad (1,0), el algoritmo A-DWRR distribuye proporcionalmente el tiempo de uso de los cores rápidos entre todas las aplicaciones. Por el contrario, Prop-SP y ACFS otorgan una fracción mayor de ciclos de core rápido a *gamess*, *bzip2* y *h264ref*, lo que permite que estas aplicaciones alcancen un mayor *speedup* al usar los cores rápidos que *gobmk*. A medida que aumentamos la prioridad de la aplicación *HPA* todos los algoritmos reducen el tiempo de ejecución de esta aplicación. Esto se debe a que el incremento gradual de la prioridad hace que aumente la fracción de tiempo de core rápido asignado a la aplicación *HPA*. Como se observa en la figura 6.14, ACFS es el único algoritmo capaz de mantener valores bajos de *unfairness* a medida que aumenta la prioridad de HPA. Por el contrario, Prop-SP y A-DWRR están sujetos a una mayor degradación de la justicia.

6.4.3. Compromiso rendimiento-justicia en ACFS

El algoritmo ACFS permite al usuario ajustar el compromiso rendimiento-justicia en un sistema AMP. Para ello, ACFS está provisto de un parámetro de configuración que llamamos *unfairness_factor* (*UF*). La figura 6.15 muestra cómo la elección del *UF* afecta al rendimiento y

la justicia de tres cargas de trabajo seleccionadas de la tabla 6.2, ejecutadas sobre el prototipo Intel QuickIA. Los valores de ambas métricas están normalizados al intervalo $(0,1)$, utilizando el mecanismo descrito en 5.2.

Los resultados revelan que el valor por defecto para *UF* (1,0), el más bajo posible, proporciona los mejores valores de *unfairness*, mientras que los valores de *UF* más altos permiten incrementar el rendimiento global a expensas de degradar la justicia. Nuestros resultados también muestran que al incrementar gradualmente el valor de *UF*, el comportamiento del planificador ACFS se acerca más al de la estrategia HSP, que optimiza el rendimiento global.

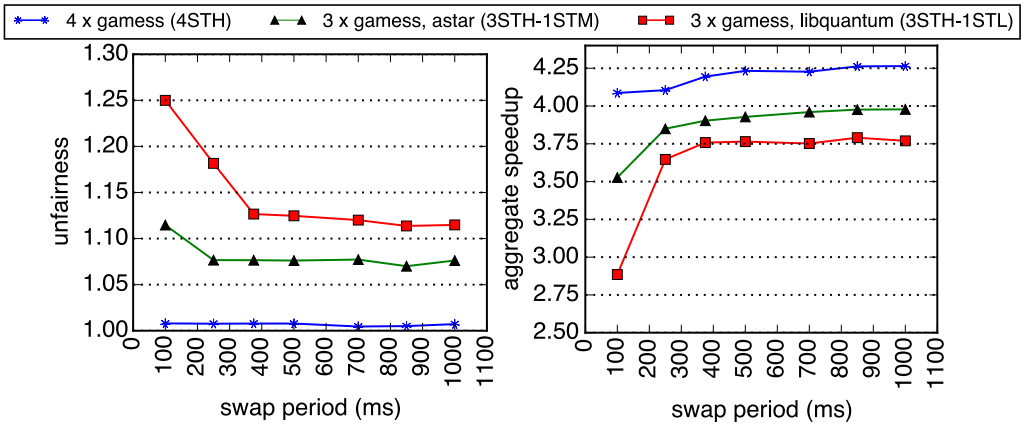
6.5. Impacto de la frecuencia de intercambio de hilos en el rendimiento global y justicia

Todos los algoritmos de planificación considerados en nuestro estudio realizan intercambios de hilos cada cierto tiempo para lograr distintos objetivos. Nuestras implementaciones de los diferentes algoritmos sobre el kernel Linux emplean el mismo mecanismo de bajo nivel para llevar a cabo esto intercambios.

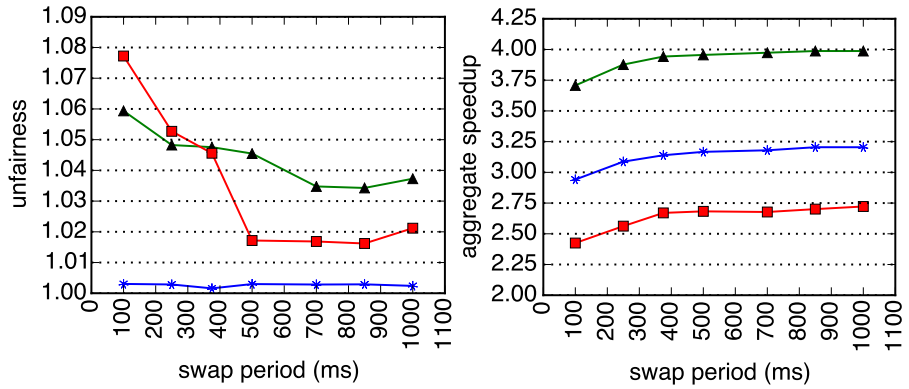
En esta sección analizamos el impacto de la frecuencia de intercambio de hilos en el rendimiento global y justicia bajo ACFS. Para llevar a cabo nuestro análisis, utilizamos 3 cargas de trabajo, con 4 programas secuenciales cada una, que se ejecutan sobre dos configuraciones AMP (plataforma Intel y plataforma AMD) formadas por 2 cores rápidos y 2 cores lentos (2F-2S). Al seleccionar las cargas de trabajo para este estudio garantizamos que la categoría de *SF* (H, M o L) de las aplicaciones que las componen no varíen en las dos plataformas consideradas. Cabe destacar que las combinaciones de aplicaciones de la tabla 6.2 no cumplen esta restricción, por lo que fue necesario construir cargas nuevas.

Las migraciones de hilos pueden introducir *overhead* tanto de *software* como *hardware* [35]. El *overhead* relacionado al *software* incluye el tiempo de mover un hilo de un core a otro. Esto implica adquirir simultáneamente los *locks* asociados a las *run queues* de las CPUs origen y destino de la migración. Nótese que en el escenario considerado (un hilo por core) las dos migraciones necesarias para el intercambio de hilos se deben serializar con frecuencia, dado que además de adquirir los *locks*, el planificador debe expropiar los dos hilos de sus respectivas CPUs. Serializar las migraciones de un intercambio puede ocasionar desequilibrios en la carga durante breves períodos de tiempo. En lo que respecta al overhead relacionado al *hardware* de una migración, se deben tener en cuenta dos aspectos: (1) el hilo tiene que reconstruir su estado de la cache tras ser migrado (hasta tres niveles de cache en la plataforma AMD), lo que implica fallos de cache adicionales; y (2) los fallos de cache que se producen tras una migración son típicamente mas costosos en las plataformas NUMA (sistema AMD) que en las máquinas UMA (Intel QuickIA) [35].

La figura 6.16 muestra el impacto en el *unfairness* y en el *ASP* al variar el período de intercambio de hilos promedio de 100ms a 1s. Los resultados revelan que la degradación en



(a) Intel QuickIA



(b) AMD platform

Figura 6.16: Unfairness y ASP al variar el período de intercambio de hilos promedio bajo ACFS sobre dos configuraciones 2F-2S.

unfairness y rendimiento global claramente decrece al incrementar el período de intercambio. En los experimentos de las secciones previas establecimos un período de intercambio de hilos promedio de 850ms, ya que esta configuración proporciona resultados satisfactorios en las plataformas utilizadas. Para la primera carga de trabajo, que consta de 4 instancias de *gamess*, observamos que el *unfairness* se mantiene sin variación incluso para valores bajos del período de intercambio de hilos. Esto se debe a que el programa *gamess* es intensivo en CPU y tiene un conjunto de trabajo pequeño; este programa experimenta un *overhead hardware* despreciable tras la migración. Sin embargo, establecer el período de intercambio de hilos promedio a un valor por debajo de 300ms introduce un *overhead software* significativo, asociado principalmente a situaciones de desbalance transitorio en la carga de las CPU„ lo cual tiene un impacto negativo en el rendimiento global. Las otras dos cargas de trabajo combinan 3 instancias de *gamess* con una aplicación medianamente intensiva en memoria (*astar*) y un programa muy intensivo en memoria (*libquantum*), respectivamente. Esas aplicaciones intensivas en memoria experimentan un *overhead hardware* significativo al ser migradas con mucha frecuencia, a diferencia de *gamess*. Esto lleva a degradaciones de rendimiento muy dispares entre aplicaciones, y en consecuencia a mayor *unfairness* para valores bajos del

período de intercambio de hilos.

6.6. Resumen del capítulo y conclusiones

En este capítulo presentamos el algoritmo de planificación ACFS, que intenta aproximar el comportamiento del planificador teórico *Opt-Unfairness*, que optimiza la justicia en un AMP. ACFS utiliza un mecanismo para monitorizar el progreso relativo realizado por los distintos hilos de una carga de trabajo, que se basa en mantener actualizados contadores de progreso por cada hilo cada *tick* de reloj. Para garantizar la justicia entre aplicaciones, ACFS intenta que los contadores de progreso de los hilos tengan valores muy similares, para lo cual es necesario migrar hilos cada cierto tiempo entre los distintos tipos de core. ACFS también permite al usuario establecer la prioridad de cada aplicación, y además integra el soporte específico para aplicaciones multi-hilo del planificador Prop-SP, presentado en el capítulo 4. Otro aspecto importante de ACFS, es que está equipado con un parámetro de configuración, denominado *Unfairness Factor* o *UF*, que permite incrementar gradualmente el rendimiento global del sistema en escenarios donde los requisitos de justicia son menos estrictos.

En este capítulo utilizamos el prototipo QuickIA de Intel para nuestra evaluación experimental. Este prototipo integra cores de alto rendimiento con pipeline de ejecución fuera de orden (Intel Xeon E5450), y cores de bajo consumo con ejecución en orden (Intel Atom). Al experimentar sobre esta plataforma asimétrica, detectamos un importante desafío: la obtención de modelos de estimación precisos para aproximar el *speedup factor* (*SF*) de los hilos de ejecución (que necesita el planificador para funcionar) en casos donde los distintos tipos cores presentan diferencias muy profundas (microarquitectura, tamaños de caché, etc.), además de la frecuencia del procesador considerada únicamente en nuestros estudios previos. En particular, observamos que las estrategias propuestas previamente para construir este tipo de modelos de estimación no resultan efectivas en el prototipo QuickIA. Para superar este problema, y poder evaluar la efectividad de ACFS de forma adecuada sobre el QuickIA, en este capítulo propusimos una nueva metodología para guiar el proceso de construcción de modelos de estimación de *SF* para AMPs. La metodología propuesta, denominada *Phase-SF*, se basa en extraer la información de caracterización del rendimiento de las distintas fases de ejecución de las aplicaciones, y a partir de esta información, generar modelos de estimación automáticamente empleando un motor de regresión aditiva.

Finalmente, para evaluar la eficacia de ACFS, realizamos una comparación exhaustiva con diversos algoritmos de planificación del estado del arte para AMPs. Nuestra evaluación experimental revela que ACFS es capaz de obtener mayores reducciones en la métrica *unfairness* que el resto de algoritmos orientados a justicia (RR [8], A-DWRR [33] y EQP [11]), para una amplia gama de cargas de trabajo que incluyen aplicaciones secuenciales y multi-hilo. Al mismo tiempo, ACFS obtiene un rendimiento global superior al de estas estrategias para la mayoría de las cargas de trabajo. Los resultados también revelan que ACFS mejora considerablemente a nuestra primera propuesta de planificación, el algoritmo Prop-SP.

7 Mejorando la eficiencia energética en AMPs

En los capítulos anteriores de esta tesis doctoral, analizamos la efectividad de las estrategias de planificación propuestas para optimizar justicia, prestando especial atención a ACFS, que además permite ajustar el compromiso entre justicia y rendimiento global. Nuestro análisis revela que para la mayoría de las cargas de trabajo, no es posible optimizar simultáneamente la justicia y del rendimiento global en AMPs, y en general es necesario realizar un reparto muy diferente de los ciclos de core rápido entre aplicaciones dependiendo del objetivo que se desee optimizar.

Investigaciones previas [65] han demostrado que a pesar de las ventajas de los multicore asimétricos en cuanto a eficiencia energética, maximizar el rendimiento global no siempre garantiza un uso efectivo de los distintos tipos de cores desde el punto de vista del consumo energético. En particular, usar los cores rápidos para ejecutar código que obtenga beneficios significativos de estos (aplicaciones con alto *speedup*) puede llevar a incrementar el consumo energético de la plataforma de manera significativa. Basándose en esto, otros investigadores han propuesto recientemente estrategias de planificación conscientes de la energía (*energy aware*) para sistemas de propósito general [65] y para sistemas empotrados [46].

A pesar de los esfuerzos destinados al diseño de estrategias de planificación conscientes de la asimetría y con la capacidad de optimizar una única métrica, hasta la fecha no se ha realizado ningún estudio que muestre la interrelación entre la eficiencia energética, el rendimiento global y la justicia. En este capítulo realizamos un estudio analítico y experimental que muestra como la optimización de uno de estos aspectos por separado afecta a los otros dos.

Para realizar nuestro estudio analítico extendemos el modelo teórico del capítulo 5 con la capacidad de aproximar el producto energía-retardo (EDP) para cargas de trabajo sintéticas. Utilizando el nuevo modelo, y un simulador basado en él, aproximamos la planificación que optimiza el *EDP* en distintos escenarios. Nuestro estudio demuestra que esta planificación óptima en cuanto a eficiencia energética se alcanza a costa de degradar el rendimiento global y

la justicia de forma significativa en muchos casos. Gracias a este análisis teórico, hallamos una métrica clave denominada *factor de eficiencia energética* o *EFF* (*Energy-Efficiency Factor*) de un hilo. Tener en cuenta este factor, permite al planificador del sistema reducir significativamente el *EDP* en cargas de trabajo multiprogramadas sobre un multicore asimétrico.

En este capítulo, también describimos y evaluamos los algoritmos de planificación EEF-Driven y ACFS-E, propuestas clave de la tesis que se basan en el *factor de eficiencia energética*. EEF-Driven aproxima la planificación teórica que optimiza (minimiza) el *EDP*. En nuestro estudio experimental, que demuestra las principales conclusiones del análisis teórico, también mostramos que EEF-Driven reduce el EDP hasta en un 15 % e incrementa el rendimiento global hasta en un 20 % con respecto a PRIM [65], el planificador *energy aware* de referencia hasta la fecha. El segundo planificador propuesto (ACFS-E) es una variante de ACFS, el algoritmo orientado a justicia descrito en el capítulo anterior. ACFS-E constituye la primera propuesta de planificación que, con ún único algoritmo, permite al administrador del sistema seleccionar la métrica que desea optimizar: justicia, eficiencia energética o rendimiento global. Para hacer esto posible, ACFS-E está equipado con dos parámetros configurables, que permiten controlar la política de prioridades dinámicas aplicada internamente por el planificador.

Para la evaluación de EEF-Driven y ACFS en un sistema real, implementamos ambos planificadores en el kernel Linux. El principal desafío asociado a la implementación de estas estrategias es el hecho de que ambos algoritmos requieren que el sistema operativo determine en tiempo de ejecución el factor de eficiencia energética (*EEF*) de un hilo. Lamentablemente, y por cuestiones técnicas, la medición directa del *EEF* en hardware asimétrico actual no es posible. Para superar esta limitación, nuestra implementación de EEF-Driven y ACFS-E utiliza modelos de estimación de *EEF* basados en contadores hardware, cuyo proceso de construcción detallamos en este capítulo.

Comenzamos este capítulo presentando el modelo teórico extendido y mostramos los resultados de nuestro estudio analítico. A continuación describimos el diseño e implementación de los planificadores EEF-Driven y ACFS-E. Para finalizar, llevamos a cabo nuestro análisis experimental.

7.1. Modelo analítico de rendimiento, justicia y eficiencia energética

Para mostrar la relación entre la justicia, rendimiento global y eficiencia energética, llevamos a cabo un estudio teórico donde evaluamos la efectividad de diferentes algoritmos de planificación. En este estudio teórico utilizamos varias cargas de trabajo y las analizamos sobre un AMP compuesto por dos cores rápidos y dos cores lentos.

Esta sección se divide en cuatro partes. En la primera parte describimos como están compuestas las cargas de trabajo sintéticas utilizadas para nuestro estudio teórico. La segunda parte presenta las fórmulas analíticas de rendimiento, justicia y eficiencia energética. La

Tabla 7.1: Aplicaciones sintéticas

App.	Benchmark	IPS_{fast}	IPS_{slow}	SF	EPI_{fast}	EPI_{slow}
A1	art	0.60	0.24	2.47	1.59	1.86
A2	astar	0.58	0.31	1.86	1.40	1.10
A3	bzip2	1.49	0.73	2.02	0.61	0.45
A4	equake	0.80	0.26	3.07	1.31	1.45
A5	galgel	1.30	0.41	3.16	0.82	0.91
A6	gamess	2.01	0.69	2.91	0.51	0.49
A7	gobmk	1.09	0.61	1.79	0.75	0.54
A8	gzip	1.07	0.63	1.70	0.78	0.56
A9	h264ref	1.83	0.93	1.96	0.51	0.37
A10	hmmer	2.80	1.04	2.69	0.42	0.36
A11	mcf	0.18	0.09	2.02	3.98	4.44
A12	mgrid	1.28	0.59	2.17	0.85	0.65
A13	perlbench	1.42	0.71	2.01	0.60	0.47
A14	perlbmk	1.77	0.78	2.27	0.54	0.41
A15	povray	1.15	0.53	2.19	0.85	0.66
A16	soplex	0.52	0.20	2.53	1.68	1.86
A17	swim	0.25	0.11	2.24	3.11	3.34
A18	vortex	1.73	0.72	2.41	0.56	0.45
A19	wupwise	1.63	0.64	2.56	0.66	0.54

Tabla 7.2: Cargas de trabajo

Carga de trabajo	Aplicaciones
W1	A5,A4,A6,A10
W2	A16,A17,A13,A9
W3	A16,A1,A18,A14
W4	A5,A4,A10,A15
W5	A5,A4,A6,A12
W6	A1,A12,A3,A13
W7	A1,A17,A7,A8
W8	A15,A11,A13,A2
W9	A4,A11,A3,A8
W10	A10,A19,A16,A9

tercera parte describe los algoritmos de planificación utilizados en nuestro análisis. En la cuarta y última parte discutimos los resultados obtenidos.

7.1.1. Cargas de trabajo

Cada carga de trabajo sintética está formada por cuatro aplicaciones secuenciales. En este escenario asumimos que las aplicaciones presentan, en cada tipo de core y durante toda su ejecución, ratios de rendimiento *fast-to-slow* constantes (que se corresponden con el *SF* de su único hilo en ejecución) y ratios de energía por instrucción (EPI) constantes. Para construir las cargas de trabajo sintéticas, analizamos el comportamiento de varias aplicaciones pertenecientes a la suite de benchmarks SPEC CPU2000 y SPEC CPU2006 cuando se ejecutan sobre la placa ARM Juno [5]. De cada benchmarks identificamos fases de programa que exhiben ratios estables de IPC y EPI en ambos tipos de core. Para obtener los valores de IPC y EPI utilizamos los contadores hardware y los registros de energía integrados en la placa ARM Juno. La tabla 7.1 muestras las propiedades de las distintas aplicaciones sintéticas. Para cada aplicación mostramos los valores de *SF*, y los valores de IPC y EPI para cada tipo de core.

Para medir los valores de EPI sobre ARM Juno consideramos el consumo de energía correspondiente al cluster de cores (rápido o lento) y el consumo neto de DRAM, cuando la aplicación corre sola en el sistema. Tuvimos en cuenta estos dos componentes de la plataforma dado que las decisiones de planificación pueden afectar tanto el consumo de energía del cluster de cores como el consumo en DRAM.

7.1.2. Fórmulas analíticas

Nuestro objetivo es encontrar la planificación teórica que optimice el *EDP* (métrica definida en la sección 3.1) para las distintas combinaciones de aplicaciones antes mencionadas. Asimismo evaluamos el impacto que esta planificación tiene en el rendimiento global y la justicia. Para esto utilizamos las ecuaciones de la sección 5.1 que dependen del SF_i de una

aplicación i y de la fracción de instrucciones (f_i) que la aplicación i completa en cores rápidos bajo un algoritmo de planificación particular. Además, para hacer el problema más tratable, hacemos las siguientes suposiciones:

- El número de aplicaciones no excede el número de cores en el AMP.
- Todas las aplicaciones de la carga de trabajo se ejecutan continuamente durante un tiempo dado CT.
- El planificador intenta mantener los cores rápidos tan ocupados como sea posible.
- No se tiene en cuenta el *overhead* debido a la contención en los recursos compartidos ni tampoco por las migraciones de hilos. Cabe destacar que estos aspectos si se tienen en cuenta en nuestro análisis experimental.

A continuación derivamos un conjunto de fórmulas para aproximar el *EDP* analíticamente en el escenario de cargas sintéticas. Antes ampliamos la notación introducida en el capítulo 5 con la siguiente notación:

- $EC_{sched,i}$: consumo de energía de una aplicación i cuando se ejecuta CT segundos bajo un algoritmo de planificación determinado.
- $EPI_{sched,i}$: ratio de energía por instrucción de una aplicación i cuando se ejecuta bajo un algoritmo de planificación determinado.
- $IPS_{sched,i}$: número de instrucciones por segundo alcanzados por la aplicación i bajo un algoritmo de planificación determinado.
- NI_i: número total de instrucciones que la aplicación i completa en CT segundos bajo un algoritmo de planificación determinado.
- F_i: fracción de tiempo (sobre CT_{sched}) que una aplicación completa bajo un algoritmo de planificación determinado.
- f_i: fracción de instrucciones sobre el total NI_i que la aplicación i completa bajo un algoritmo de planificación determinado.

Realizamos la derivación del *EDP* analítico como sigue:

$$EDP = \frac{Energia\_consumida \cdot CT}{Total\_instrucciones\_retiradas}$$

$$= \frac{Energia\_consumida}{IPS}$$

$$= \frac{\sum_{i=1}^{n} EC_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}} = \frac{\sum_{i=1}^{n} NI_i \cdot EPI_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}} \tag{7.1}$$

$$= \frac{\sum_{i=1}^{n} CT \cdot IPS_{sched,i} \cdot EPI_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}}$$

$$= CT \cdot \frac{\sum_{i=1}^{n} IPS_{sched,i} \cdot EPI_{sched,i}}{\sum_{i=1}^{n} IPS_{sched,i}}$$

A su vez, $IPS_{sched,i}$ y $EPI_{sched,i}$ pueden definirse en función de varias propiedades de la aplicación (que se muestran en la tabla 7.1), y en función de f_i. Usando las siguientes fórmulas:

$$IPS_{sched,i} = \frac{IPS_{fast}}{f_i + SF_i \cdot (1 - f_i)} \tag{7.2}$$

$$EPI_{sched,i} = EPI_{fast} \cdot f_i + EPI_{slow} \cdot (1 - f_i) \tag{7.3}$$

7.1.3. Algoritmos de planificación analizados

Ahora analizaremos los valores de las métricas *ASP*, *unfairness* y *EDP* al ejecutar varias cargas de trabajo sintéticas bajo cuatro estrategias de planificación para AMP:

- HSP, un planificador que optimiza el rendimiento asignando a cores rápidos aquellas aplicaciones de la carga de trabajo con mayor *speedup*. Para estas aplicaciones $F_{app} = 1$. Las aplicaciones restantes se asignan a cores lentos. Para estas aplicaciones $F_{app} = 0$.
- *Opt-Unf*, un planificador teórico que asegura el óptimo *unfairness* para el máximo valor de *ASP* alcanzable.
- *Opt-EDP*, un planificador teórico que asegura el óptimo *EDP* (valor más bajo) para el máximo valor de *ASP* alcanzable.

- EEF-Driven, uno de los planificadores propuestos en este capítulo. En particular, EEF-Driven asigna a cores rápidos las N_{FC} aplicaciones que alcanzan el valor más alto de *Energy-Efficiency Factor*, el resto de las aplicaciones se asignan a cores lentos.

El *Energy-Efficiency Factor* (*EEF*) se define como:

$$EEF = \frac{SF}{EPI_{fast}} \tag{7.4}$$

Donde EPI_{fast} está dado en *nanojoules*.

Llegamos a este factor a partir del análisis de la asignación de hilos a cores realizada por el planificador *Opt-EDP*. En particular, EEF-Driven utiliza *EEF* para aproximar el comportamiento de este planificador teórico.

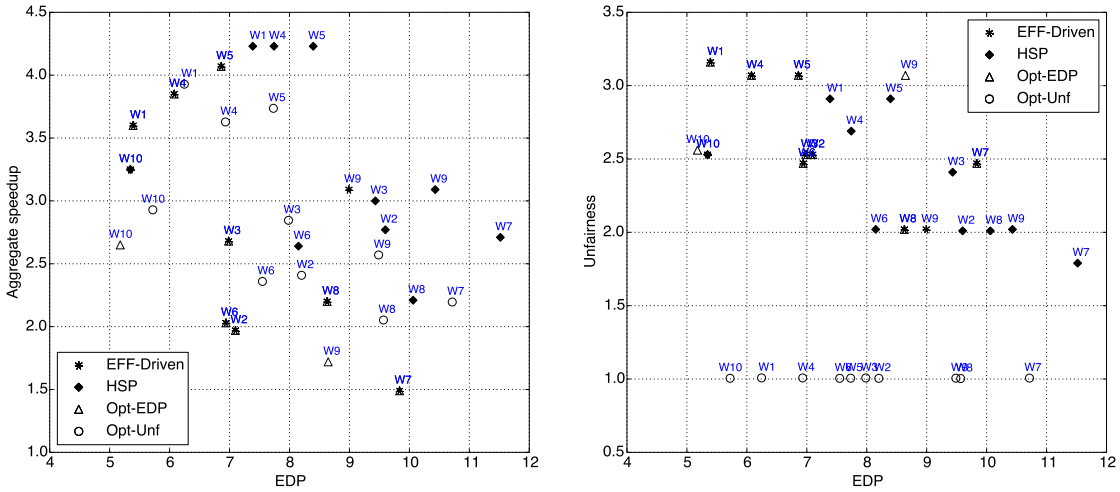
Como mencionamos en el capítulo 5, para determinar la distribución de ciclos de core rápido por aplicación bajo el planificador teórico *Opt-Unf* es necesario efectuar una exploración exhaustiva del espacio de búsqueda. Lo mismo ocurre con el planificador teórico *Opt-EDP*. En este capítulo empleamos el mismo simulador del capítulo 5 que hace uso de las ecuaciones 5.7, 5.6 y 7.1 y busca la solución óptima en cada caso a través de un algoritmo *branch-and-bound*. Bajo *Opt-EDP* es necesario tener en cuenta los valores de *IPS*, *EPI* y *SF* de cada aplicación para permitir aproximar el *EDP* de las distintas soluciones exploradas. En este caso, dada una carga de trabajo, el algoritmo de búsqueda calcula el par (*ASP*, *EDP*) para cada posible distribución de los ciclos de core rápido entre las aplicaciones. Las soluciones candidatas se crean variando F_i de 0 a 1 en pasos de 0,01, de manera que $\sum_{i=1}^{n} F_i = N_{FC}$. [1]

Nótese que para determinar la planificación óptima, asumimos que la frecuencia de cada cluster de cores (rápido o lento) se establece a la configuración por defecto, y no se modifica durante la ejecución. Reducir la frecuencia del procesador puede ayudar a reducir el consumo de energía en AMPs [2], pero esto puede traducirse en degradación del rendimiento de las aplicaciones, lo cual puede impactar negativamente en el rendimiento global y en la justicia. Analizar los efectos de la planificación consciente de la asimetría en combinación con políticas de DVFS (*Dynamic Voltage and Frequency Scaling*) está fuera del alcance de esta tesis, pero constituye una línea interesante de trabajo futuro.

7.1.4. Análisis de los resultados

Para evaluar la efectividad de los planificadores construimos distintas cargas de trabajo (tabla 7.2) compuestas por combinaciones de las aplicaciones sintéticas de la tabla 7.1. Es importante destacar que al seleccionar el conjunto de cargas de trabajo para nuestra evaluación descartamos aquellas donde HSP y *Opt-EDP* realizan la misma distribución de ciclos de core

[1] Aproximamos *ASP* y *EDP* en función de F_i en lugar de f_i, la equivalencia viene dada por la ecuación 5.5



(a) EDP vs.ASP

(b) EDP vs. Unfairness

Figura 7.1: Valores de ASP, EDP y unfairness para las cargas de trabajo de la tabla 7.2 bajo los distintos planificadores. En la figura de la izquierda, los mejores resultados se encuentran más próximos a la esquina superior izquierda. En la figura de la derecha, los mejores resultados se encuentran más próximos a la esquina inferior izquierda.

rápido. Para estas cargas descartadas, ambos planificadores obtienen los mismos valores de *ASP*, *unfairness* y *EDP*.

Los resultados obtenidos con el simulador se muestran en la figura 7.1a (relación entre *ASP* y *EDP*) y en la figura 7.1b (relación entre *unfairness* y *EDP*). HSP alcanza los valores más altos de *ASP* (rendimiento global) a expensas de obtener valores de *EDP* mayores que el óptimo (incremento de hasta un 22 %). Por el contrario, *Opt-EDP* alcanza el menor valor de *EDP* para todas las cargas de trabajo pero a costa de una importante degradación del rendimiento en algunos casos (de hasta un 44 % para W9). HSP y Opt-EDP son inherentemente injustos. Ambos planificadores logran beneficios en cuanto a rendimiento y eficiencia energética, respectivamente, pero degradando significativamente la justicia (valores considerablemente mayores que 1).

En cuanto a EEF-Driven, observamos que en la mayoría de los casos realiza la misma distribución de ciclos de core rápido que *Opt-EDP*. En los escenarios donde EEF-Driven y *Opt-EDP* realizan una distribución diferente (W9 y W10), EEF-Driven sufre un leve incremento en EDP (hasta un 4 %) mientras que alcanza una mejora significativa en rendimiento (hasta un 80 %). Los resultados sugieren que EEF-Driven constituye una buena aproximación a *Opt-EDP* (EEF-Driven alcanza el óptimo *EDP* para el máximo valor de *ASP* alcanzable).

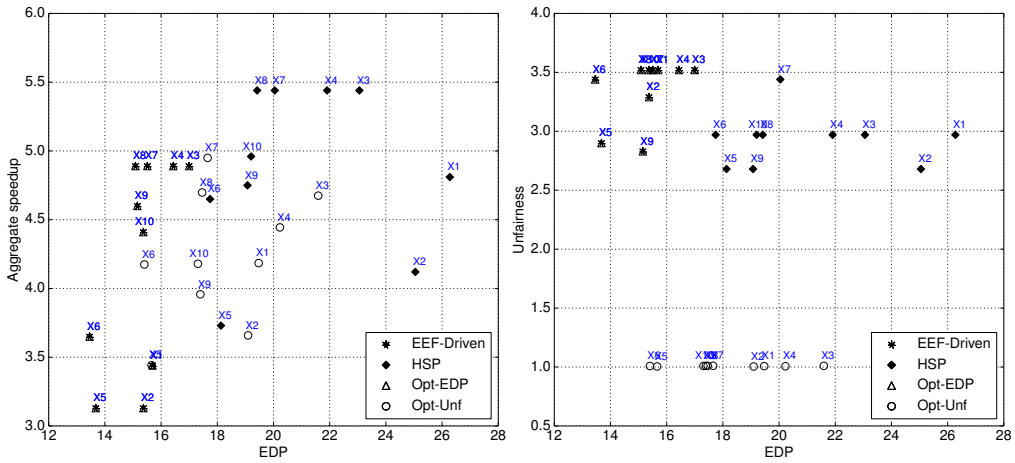
Para verificar que las conclusiones obtenidas hasta el momento son también válidas para otras plataformas AMP, experimentamos con la placa de desarrollo Odroid XU4 [22]. Este sistema cuenta con un procesador big.LITTLE de 32 bits que combina cores lentos Cortex A7 y cores rápidos Cortex A15. En un sistema de este tipo, los rangos de *SF* y *EPI* para los

Tabla 7.3: Aplicaciones sintéticas

Aplicación	Benchmark	IPS_{fast}	IPS_{slow}	SF	EPI_{fast}	EPI_{slow}
B1	astar	0.76	0.37	2.05	4.53	1.70
B2	crafty	2.46	1.00	2.47	1.54	0.54
B3	equake	0.54	0.19	2.83	6.32	3.30
B4	gobmk	1.68	0.76	2.21	2.02	0.69
B5	h264ref	3.08	1.04	2.97	1.31	0.57
B6	mcf	0.29	0.08	3.52	10.57	7.94
B7	mgrid	2.53	0.74	3.44	1.86	0.94
B8	perlbench	2.45	1.00	2.45	1.46	0.51
B9	perlbmk	2.80	1.04	2.68	1.36	0.54
B10	povray	2.15	0.74	2.90	1.91	0.62
B11	sixtrack	1.90	0.48	3.92	1.90	1.01
B12	soplex	0.62	0.19	3.29	5.71	3.46
B13	swim	0.75	0.23	3.21	4.89	2.93

Tabla 7.4: Cargas de trabajo

Workload	Applications
X1	B6,B12,B5,B2
X2	B12,B3,B9,B8
X3	B11,B6,B5,B3
X4	B11,B6,B5,B1
X5	B10,B3,B9,B8
X6	B7,B13,B5,B9
X7	B11,B6,B7,B5
X8	B11,B6,B5,B10
X9	B11,B3,B9,B4
X10	B6,B7,B5,B4



(a) EDP vs ASP

(b) EDP vs Unfairness

Tabla 7.5: Valores de ASP, EDP y unfairness para las cargas de la tabla 7.4.

benchmarks SPEC CPU son muy diferentes a los observados en el procesador big.LITTLE de la placa Juno. Esto se debe al hecho de que los cores del mismo tipo (es decir, rápidos o lentos) en estas plataformas difieren en características microarquitectónicas, en la frecuencia del procesador y en consumo de potencia [4, 22]. En particular, observamos que el consumo de potencia de un core rápido del procesador de 32 bits (Odroid XU4) es aproximadamente cuatro veces superior al de un core rápido del procesador de 64 bits (Juno). En estas circunstancias, para cubrir un amplio espectro de perfiles de rendimiento y consumo de energía para la placa Odroid, tuvimos que seleccionar un conjunto específico de ejecución representativas para este sistema. Para esto, ejecutamos diferentes benchmarks de SPEC CPU solos en ambos tipos de core e identificamos fases de ejecución con un valor estable de *IPS* y *EPI*. Nótese que, a diferencia de la placa Juno, la Odroid XU4 no cuenta con registros de energía integrados en la plataforma. Para medir el consumo de energía en este sistema, utilizamos un monitor externo de consumo energético (Odroid Smart Power), a partir del cual pueden obtenerse datos empleando la herramienta PMCTrack [55].

A partir de la información a nivel de fase recopilada de la ejecución de los distintos benchmarks SPEC CPU en la placa Odroid XU4, definimos un conjunto de aplicaciones

sintéticas cuyas propiedades se muestran en la tabla 7.3. Luego, utilizando estas aplicaciones sintéticas construimos combinaciones diferentes de 4 aplicaciones cada una, siguiendo el mismo procedimiento utilizado para la placa Juno. La tabla 7.4 muestra un subconjunto representativo de las cargas de trabajo que exploramos para un sistema AMP hipotético compuesto por dos cores rápidos y dos cores lentos. Los resultados asociados a estas cargas de trabajo se muestran en las figuras 7.5a y 7.5b. Estos resultados muestran tendencias similares a las observadas para las cargas de trabajo sintéticas en el procesador big.LITTLE de 64 bits. Una vez más, el planificador HSP es capaz de proporcionar los mejores valores de rendimiento a expensas de una degradación significativa de EDP (hasta un 65 %). También observamos que la degradación del *EDP* con respecto a *Opt-EDP* es incluso mayor en este sistema, debido al mayor rango en los valores de *EPI* observados en esta plataforma. Por otra parte, los resultados revelan que en este escenario, EEF-Driven y *Opt-EDP* realizan la misma distribución de ciclos de core rápido para todas las cargas de trabajo, por lo que EEF-Driven siempre alcanza el valor máximo de *ASP* para el óptimo *EDP*. Lo que es aún más importante es el hecho de que en este escenario no encontramos ninguna carga de trabajo donde EEF-Driven y *Opt-EDP* se comporten de manera diferente.

La principal conclusión que obtenemos de estos resultados es que la justicia, el rendimiento global y la eficiencia energética constituyen objetivos contrapuestos, ya que cualquier intento de optimizar una de las métricas lleva a degradar las otras sustancialmente. Más concretamente, obtener valores aceptables de justicia (valores de *unfairness* cercanos a 1) generalmente implica una degradación muy significativa en el rendimiento global y en la eficiencia energética.

7.2. Diseño de los algoritmos propuestos

El estudio teórico presentado en la sección anterior abre algunos interrogantes:

- ¿Cómo puede implementarse el planificador EEF-Driven sobre un sistema real y cómo puede reaccionar a los cambios de fase un programa en tiempo de ejecución?
- ¿Cómo podemos determinar el *EEF* de una aplicación en tiempo de ejecución?
- ¿Es posible incluir un parámetro de configuración en el planificador para controlar el rendimiento global, la justicia y el grado de eficiencia energética?

En las siguientes subsecciones proporcionamos una respuesta a cada uno de estos interrogantes.

7.2.1. El planificador EEF-Driven

El algoritmo de planificación EEE-Driven asigna hilos a los distintos tipos de core preservando el balance de carga en AMP, y realiza periódicamente intercambios de hilos entre los

cores en función del *EEF* de cada hilo.

Cuando un nuevo hilo entra al sistema, EEF-Driven aún desconoce el*EEF* de ese hilo por lo que le asigna un valor de *EEF* por defecto. En nuestra implementación optamos por utilizar como valor por defecto el valor más bajo de *EEF* observado para los benchmarks de la suite SPEC CPU sobre la plataforma utilizada. De esta forma, los hilos con un valor bajo de *EEF* y legítimamente asignados a cores rápidos, no serán migrados a cores lentos cuando un nuevo hilo entra al sistema.

Inicialmente los hilos se asignan a los distintos cores para preservar el balance de carga en el sistema. A medida que los hilos comienzan su ejecución, el planificador utiliza los contadores hardware del procesador para obtener distintas métricas de rendimiento, que serán necesarias para estimar el valor de *EEF* de los hilos en tiempo de ejecución. Durante la ejecución de una carga de trabajo pueden darse dos situaciones: 1) un nuevo hilo entra al sistema 2) el *EEF* de un hilo ya existente varía debido a que éste entra en una nueva fase de ejecución. En función del *EEF* de los hilos, EEF-Driven debe verificar si es necesario realizar una nueva asignación de hilos a cores. Si este es el caso, el algoritmo debe realizar migraciones de hilos. Las migraciones deben asegurar lo siguiente:

1. Todos los hilos corriendo en los cores rápidos deben tener un valor de *EEF* más alto que el hilo con mayor valor de *EEF* corriendo en los cores lentos.

2. El balance de carga debe preservarse.

El algoritmo de planificación debe garantizar en todo momento que el hilo con el menor valor de *EEF* corriendo en el core rápido, tiene un valor de *EEF* mayor que el del hilo con el mayor valor de *EEF* corriendo en el core lento. Sin embargo, esta situación puede alterarse si se produce un cambio en el valor de *EEF* de un hilo o cuando un hilo cambia de estado (por ejemplo, se bloquea debido a una operación de entrada-salida). En el primer caso, el algoritmo de planificación realizará un intercambio de hilos entre los distintos tipos de cores. En el segundo caso, el algoritmo de planificación sólo realizará la migración de un hilo al core más desocupado para balancear la carga.

Para facilitar la selección del hilo a migrar, EEF-Driven mantiene por cada tipo de core una lista de hilos activos ordenada por *EEF*. Por razones de eficiencia, la lista de los cores rápidos está ordenada de forma ascendente, mientras que la lista de cores lentos está ordenada de forma descendente. Como resultado, la búsqueda del mejor candidato a migrar tiene complejidad constante.

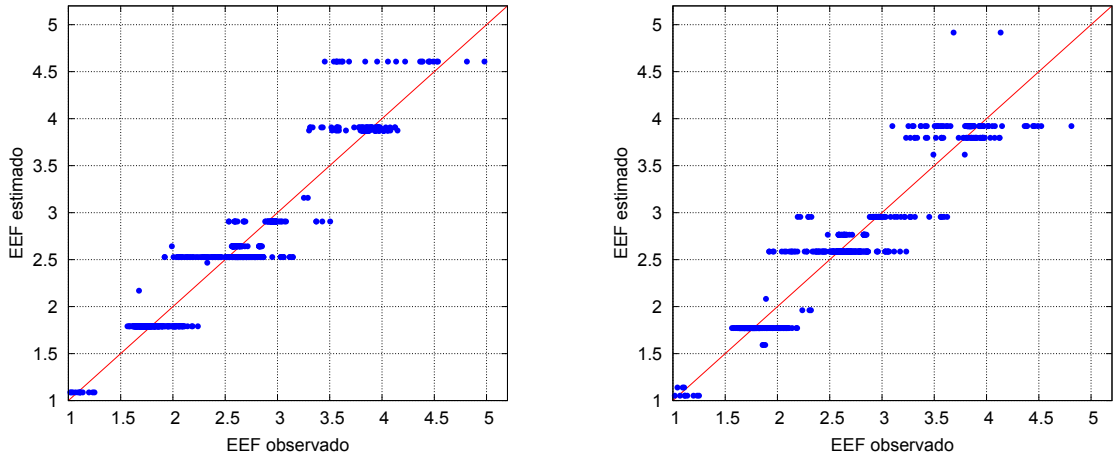
Cuando cambia el *EEF* de un hilo, la lista por *EEF* del core donde el hilo se encuentra en ejecución debe reorganizarse. No obstante, observamos que si las estimaciones de *EEF* se actualizan durante cambios de fase abruptos es posible que el planificador realice migraciones frecuentes y potencialmente costosas, que incluso podrían ser innecesarias si la nueva fase de

ejecución es de muy corta duración. Para reducir el número de operaciones de reorganización de listas y el número de migraciones innecesarias utilizamos un mecanismo de detección de fases que intenta capturar fases de *EEF* de grano grueso. De esta forma, las estimaciones de *EEF* se actualizan cuando un hilo ingresa en una fase estable. Para lograr esto, mantenemos la *media móvil* de los valores de *EEF* obtenidos a lo largo del tiempo. El valor de media móvil obtenido en un intervalo determinado se compara con el valor de media móvil del intervalo previo. Si la diferencia entre los dos valores supera un umbral determinado (12 % en nuestra plataforma experimental) entonces nos encontramos ante un cambio de fase. Dos o más intervalos de muestreo consecutivos sin transiciones de fases, indican que se está en presencia de una fase estable.

7.2.2. Determinando el EEF en tiempo de ejecución

El *EEF* de un hilo puede variar a medida que éste atraviesa distintas fases de ejecución. Para optimizar el *EDP*, el algoritmo EEF-Driven debe ajustar la asignación de hilos a cores dinámicamente en función del *EEF*. Para hacer esto posible, el planificador debe estar equipado con un mecanismo que le permita obtener el *EEF* de un hilo en tiempo de ejecución. Lamentablemente, medir *EEF* directamente no es posible en la práctica. De acuerdo a la ecuación 7.4, para calcular *EEF* es necesario obtener el *SF* y EPI_{fast}. Como vimos anteriormente, el *SF* de un hilo puede aproximarse en tiempo de ejecución mediante estimación. Sin embargo, en plataformas asimétricas comerciales no es posible obtener el valor de EPI_{fast} de un hilo cuando se ejecuta simultáneamente con otros hilos. Esto se debe principalmente a que los sistemas comerciales actuales, que cuentan con sensores o registros para medir energía, permiten obtener valores de consumo de energía del sistema en su conjunto, pero no obtener medidas de energía de hilos individuales. En la plataforma utilizada en este capítulo, la placa de desarrollo ARM Juno, podemos medir el consumo energético de cada clusters de cores (cores rápidos y lentos) y de la DRAM [6, 4]. Con este soporte, el sistema operativo no puede aislar la contribución que cada aplicación de la carga de trabajo realiza en el consumo energético global.

Una alternativa para poder determinar el *EEF* de un hilo en tiempo de ejecución es obtener un modelo de estimación que permita aproximar el *EEF* a partir de métricas de rendimiento de alto nivel de un hilo monitorizadas en tiempo de ejecución. Para la construcción de este modelo utilizamos una variante de la metodología utilizada en el capítulo 6 (*Phase-SF*). Con esta metodología ejecutamos un conjunto de benchmarks SPEC CPU2000 y CPU2006 sobre ambos tipos de core y monitorizamos distintas métricas de alto nivel usando los contadores hardware del procesador. Durante la ejecución de los benchmarks en los cores rápidos también monitorizamos EPI_{fast}. Para obtener información de los contadores hardware y de los registros de energía utilizamos la herramienta PMCTrack [55]. A partir de los datos recolectados extraemos distintas fases de ejecución de grano grueso con un valor de *EEF* estable. Finalmente, utilizamos los datos de las fases de *EEF* como entrada a WEKA para generar dos modelos de estimación de *EEF*, uno para cada tipo de core.



(a) Core rápido

(b) Core lento

Figura 7.2: Predicción de EEF sobre los cores rápidos y lentos en el sistema ARM Juno con la metodología Phase-SF. Los puntos ubicados en la diagonal representan estimaciones perfectas.

La figura 7.2 muestra la predicción de *EEF* en ambos tipos de core sobre la placa ARM Juno. Los coeficientes de correlación obtenidos sobre los cores rápidos y lentos son 0.97 y 0.95, respectivamente.

La tabla 7.6 muestra las métricas de rendimiento y el conjunto de eventos hardware de las cuales depende el modelo obtenido. Para implementar ambos modelos de estimación empleamos un módulo de monitorización de PMCTrack. Este módulo, encapsulado dentro de un modulo del kernel, provee al algoritmo de planificación de estimaciones de *EEF* para cada hilo en tiempo de ejecución.

7.2.3. El planificador ACFS-E

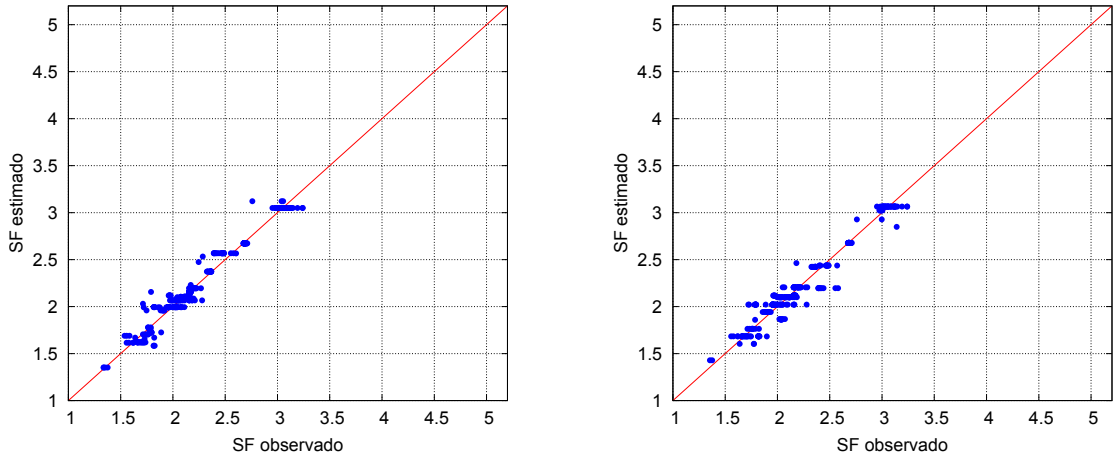
La mayoría de los algoritmos de planificación que intentan optimizar el rendimiento global u ofrecer mejoras en la eficiencia energética (como HSP y EEF-Driven, respectivamente) están sujetos a una importante limitación: proporcionan un compromiso fijo entre rendimiento, eficiencia energética y justicia. Al mismo tiempo, como revela nuestro estudio analítico, estas estrategias son inherentemente injustas. Para optimizar el rendimiento o la eficiencia energética, estos planificadores asignan un subconjunto de hilos de la carga de trabajo a los cores rápidos y relegan el resto de los hilos a los cores lentos, durante la mayor parte de su ejecución. Otra limitación de estas estrategias de planificación es que no poseen soporte a prioridades definidas por el usuario y tampoco están equipados con ningún mecanismo que permita ajustar el grado de una métrica con respecto a otra (por ejemplo: lograr mayor justicia a expensas de degradar la eficiencia energética). El algoritmo de planificación ACFS propuesto en el capítulo 6 sí soporta prioridades definidas por el usuario y está además equipado con un parámetro de configuración (*UF*) que permite ajustar el compromiso rendimiento-justicia.

Tabla 7.6: Metricas de rendimiento y eventos hardware asociados para predecir EEF y SF en ARM Juno. Optamos por dejar indicado en inglés los nombres de los eventos hardware y las métricas correspondientes para facilitar la consulta de información sobre éstos en la documentación técnica proporcionada por ARM.

Eventos Hardware	**Métricas de Rendimiento**	**EEF (fast)**	**EEF (slow)**	**SF (fast)**	**SF (slow)**
Instructions retired, processor cycles	Instructions per cycle	✓	✓	✓	✓
L2 (LLC) cache misses	LLC cache misses per 1K instr.	✓	✓	✓	✓
L2 Data cache accesses	L2 Data cache accesses per 1K instr.	✓	✓		
L1 data cache misses	L1 data cache misses per 1K instr.		✓	✓	✓
Data memory access	Data memory accesses per 1K instr.			✓	
Predictable branches speculatively executed	Predictable branch speculatively executed per 1K instr.	✓		✓	
Conditional branch executed	Conditional branch executed per 1K instr.		✓		✓
Mispredicted or not predicted branches speculatively executed	Mispredicted or not predicted branch speculatively executed per 1K instr.			✓	✓
L1 instruction TLB misses	L1 ITLB misses per 1M instr.	✓		✓	
`STALLS_A`: Counts every cycle there is an interlock that is not because of an Advanced SIMD or Floating-point instruction, and not because of a load/store instruction waiting for data to calculate the address in the AGU	`STALLS_A` per 1K cycles.		✓		✓
`STALLS_B`: Counts every cycle the DPU IQ is empty and there is an instruction cache miss being processed	`STALLS_B` per 1K cycles.		✓		✓
`STALLS_C`: Counts every cycle the DPU IQ is empty and there is an instruction micro-TLB miss being processed	`STALLS_C` per 1K cycles.				✓

Sin embargo, ACFS no tiene en cuenta la eficiencia energética.

Para superar esta limitación, proponemos el algoritmo de planificación ACFS-E, una variante del algoritmo ACFS que permite ajustar el compromiso entre el grado de eficiencia energética y la justicia mediante un parámetro de configuración llamado EDP_{factor}. El mecanismo propuesto funciona como sigue. Cuando el parámetro EDP_{factor} tiene el valor por defecto (1.0), el algoritmo se comporta como la implementación base de ACFS, es decir asegura un valor óptimo de *unfairness* para el máximo valor de *ASP* alcanzable. Cuando el valor de EDP_{factor} es mayor que el valor por defecto, el algoritmo reduce el *EDP* a expensas de degradar la justicia. Intuitivamente, para lograr esto es preciso incrementar la fracción de ciclos de core rápido otorgado a aquellas aplicaciones de la carga de trabajo con un valor mayor de *EEF*, mientras que se reduce la fracción de ciclos de core rápido para el resto de las aplicaciones. Al hacer esto se debe tener en cuenta el grado de eficiencia energética de cada hilo a la hora de incrementar su contador de progreso ($amp\_vruntime$). Por lo tanto, el



(a) Core rápido

(b) Core lento

Figura 7.3: Predicción del SF sobre los cores rápidos y lentos de la placa ARM Juno empleando los modelos generados con la metodología Phase-SF. Los puntos sobre la diagonal representan estimaciones perfectas.

incremento $\Delta amp\_vruntime$ bajo ACFS-E se define como sigue:

$$\Delta amp\_vruntime = \frac{100 \cdot W_{def}}{S_{core} \cdot W_t \cdot (1 + \frac{(EDP_{factor}-1)\cdot(EEF_t - EEFmin)}{EEF_{max}-EEF_{min}})} \tag{7.5}$$

Donde EEF_t es el *EEF* de un hilo t, EEF_{max} y EEF_{min} son el máximo y el mínimo *EEF* observado entre los hilos de la carga de trabajo, respectivamente. Intuitivamente, con la nueva definición de $\Delta amp\_vruntime$, el valor de $amp\_vruntime$ de aquellos hilos con mayor *EEF* se incrementa más lentamente que el de los hilos con menor *EEF*, lo que conlleva a una fracción mayor de ciclos de core rápido para aquellas aplicaciones con mayor *EEF*. Como resultado esto permite reducir el *EDP*. En la sección 7.3.2 analizamos el efecto del parámetro EDP_{factor} en una implementación del algoritmo de planificación ACFS-E sobre el kernel Linux.

Para calcular el valor de $\Delta amp\_vruntime$ de un hilo en un punto específico de su ejecución, ACFS-E necesita determinar el *EEF* y el *SF* de ese hilo. Para obtener el valor de *EEF* utilizamos el modelo de estimación presentado en la Sección 7.2.2.

Para la estimación del valor de *SF* derivamos dos modelos de estimación (uno para cada tipo de core) empleando la metodología *Phase-SF* (ver sección 6.3.2) sobre la placa ARM Juno. La figura 7.3 muestra la calidad de la predicción de *SF* en ambos tipos de core utilizando los modelos obtenidos. El coeficiente de correlación observado es 0.95 para la predicción en ambos tipos de core.

La tabla 7.6 muestra el conjunto de métricas de rendimiento y los eventos hardware asociados que son necesarios en cada modelo de estimación.

7.3. Evaluación experimental

En nuestro análisis experimental comparamos EEF-Driven con ACFS y con otros planificadores para AMP: HSP, PRIM [65] y RR. Asimismo, evaluamos la efectividad del parámetro de configuración *EDP_factor* del algoritmo ACFS-E. Para llevar a cabo nuestro análisis, implementamos todos los algoritmos de planificación en el kernel Linux.

A excepción del algoritmo RR, los algoritmos de planificación evaluados necesitan acceder a los contadores hardware del procesador. En particular, HSP y ACFS hacen uso de los contadores hardware en los modelos de estimación de *SF*, mientras que EEF-Driven y ACFS-E utilizan los contadores hardware en los modelos de estimación de *EEF*. El planificador PRIM emplea un conjunto de reglas específicas de la plataforma basadas en contadores hardware para determinar si un intercambio de hilos producirá o no ahorro de energía. Este algoritmo fue evaluado anteriormente mediante simulación. Para permitir comparar esta estrategia contra nuestra propuesta, creamos una implementación de PRIM en el kernel Linux, y adaptamos las reglas presentadas en [65] para el hardware asimétrico real que utilizamos. Los diferentes modelos de estimación, basados en contadores hardware, fueron implementados dentro de un módulo del kernel utilizando varios módulos de monitorización de PMCTrack [55]. De este modo, garantizamos que la implementación de los diferentes planificadores (incluidos en el kernel Linux) sea completamente independiente de la plataforma. Es importante mencionar que los módulos realizan el muestreo de los contadores cada *200ms* por hilo. Utilizando este valor observamos que el *overhead* asociado al muestreo de los contadores hardware y la estimación del *EEF* y *SF* de los hilos es despreciable.

Para evaluar la efectividad de los distintos algoritmos de planificación utilizamos la placa de desarrollo ARM Juno descrita en la sección 2.1, un sistema asimétrico comercial que permite al usuario tomar medidas de consumo energético, una característica que no poseen los sistemas asimétricos utilizados en capítulos anteriores. Sobre esta arquitectura exploramos dos configuraciones: 2F-4S, compuesta por dos cores rápidos y cuatro cores lentos, y 1F-3S, compuesta por un core rápido y tres cores lentos. En la configuración 2F-4S las aplicaciones que corren sobre los cores rápidos comparten la memoria cache de último nivel de 2MB. La configuración 1F-3S nos permitió estudiar el comportamiento de las distintas estrategias de planificación en un escenario donde las aplicaciones tienen que competir por un único core rápido con una caché de último nivel de 2MB privada.

Las cargas de trabajo evaluadas están compuestas por aplicaciones de las suites de benchmarks SPEC CPU2006 y CPU2000. Optamos por utilizar este tipo de cargas de trabajo para garantizar una comparación justa contra PRIM, HSP, RR y ACFS, ya que estos planificadores fueron evaluados anteriormente utilizando cargas de trabajo similares. En todos los experimentos el número total de hilos en la carga de trabajo coincide con el número de cores de la plataforma. Todas las aplicaciones de una carga de trabajo se envían a ejecutar simultáneamente. Cuando una aplicación termina, se vuelve a ejecutar tantas veces como sea necesario hasta que la aplicación de mayor duración del conjunto se ejecute tres veces. Luego, obtene-

Tabla 7.7: Cargas de trabajo multi-aplicación

Carga	Aplicaciones
M1	equake,soplex,vortex,perlbmk,povray,gobmk
M2	equake,gamess,hmmer,perlbench,gzip,gobmk
M3	galgel,equake,hmmer,povray,mgrid,gobmk
M4	galgel,gamess,hmmer,povray,perlbench,gobmk
M5	equake,gamess,hmmer,gobmk,crafty,sixtrack
M6	galgel,equake,gamess,hmmer,sixtrack,povray
M7	galgel,equake,hmmer,bzip2,perlbench,h264ref
M8	gamess,art,gobmk,crafty,sixtrack,vortex
M9	gamess,art,bzip2,gobmk,sixtrack,vortex
M10	soplex,bzip2,perlbench,gzip,h264ref,gobmk
M11	equake,hmmer,sixtrack,povray
M12	equake,hmmer,povray,astar
M13	galgel,hmmer,povray,mgrid
M14	equake,gamess,hmmer,perlbench
M15	equake,vortex,povray,h264ref
M16	galgel,hmmer,crafty,perlbench
M17	soplex,vortex,povray,h264ref
M18	art,vortex,perlbmk,povray
M19	art,perlbmk,povray,h264ref
M20	art,perlbmk,sixtrack,povray

mos el *ASP* y *unfairness* para el planificador en cuestión utilizando la media geométrica de los tiempos de ejecución para cada programa. Para medir el *EDP* de cada carga de trabajo (usando la ecuación 3.5) utilizamos la herramienta PMCTrack[55], que permite obtener información de monitorización de los registros de energía y los contadores de rendimiento disponibles en la placa ARM Juno.

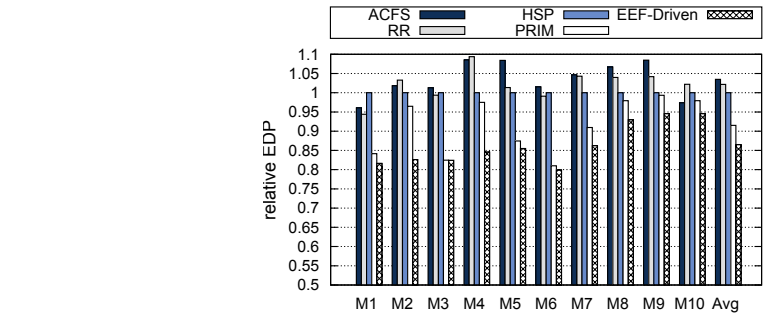
El resto de esta sección se divide en dos partes. En la primera parte (sección 7.3.1) analizamos la efectividad del algoritmo EEF-Driven. En esa sección también discutimos los resultados de la implementación base de ACFS, que se comporta como el planificador ACFS-E cuando el parámetro de configuración *EDP_factor* se establece al valor por defecto (1.0). En la segunda parte (sección 7.3.2) analizamos el impacto de variar el parámetro *EDP_factor* bajo ACFS-E.

7.3.1. Evaluación del algoritmo EEF-Driven

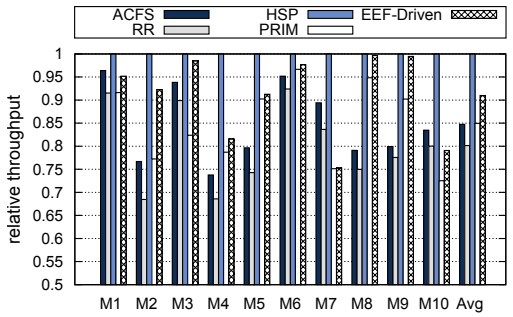
7.3.1.1. Selección de cargas de trabajo

Realizamos nuestra evaluación experimental utilizando las 20 cargas de trabajo multi-aplicación que se muestran en la tabla 7.7. Las primeras 10 cargas de trabajo (M1-M10) están compuestas por 6 aplicaciones, y se evaluaron en la configuración 2F-4S. Las 10 cargas de trabajo restantes(M11-M20) están compuestas por 4 aplicaciones, y se evaluaron sobre la configuración 1F-3S.

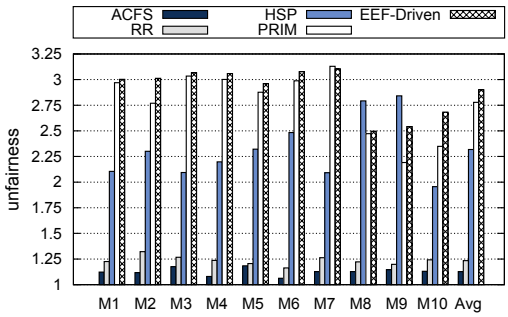
Para seleccionar las cargas de trabajo, generamos combinaciones aleatorias de programas (de 4 o 6 aplicaciones cada una). Más concretamente, para generar las combinaciones utilizamos 18 aplicaciones diferentes de las suites de benchmarks SPEC CPU2000 y CPU2006 que cubren un amplio espectro de *SF* y *EEF*. Luego, descartamos aquellas combinaciones de aplicaciones donde HSP y EEF-Driven realizan la misma distribución de ciclos de core



(a) EDP relativo



(b) ASP relativo



(c) Unfairness

Figura 7.4: Resultados para las cargas de trabajo M1-M10 bajo la configuración 2F-4S

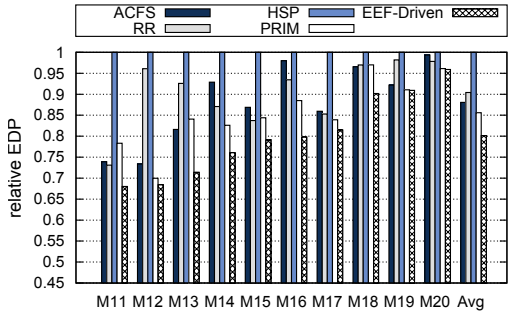
rápido durante la mayor parte de la ejecución. Las cargas de trabajo descartadas no permiten mostrar el potencial de los algoritmos que persiguen mejorar la eficiencia energética. En estas cargas de trabajo, las aplicaciones con mayor *SF* generalmente tienen mayor *EEF*, por lo tanto maximizar el rendimiento global permite obtener mejoras en la eficiencia energética. Por otro lado, bajo los algoritmos HSP y EEF-Driven, esas cargas de trabajo muestran valores muy cercanos de *ASP*, *unfairness* y *EDP*. Por el contrario, para las cargas seleccionadas estos algoritmos se comportan de forma muy diferente.

Las aplicaciones en cada carga de trabajo de la tabla 7.7 se listan en orden descendente por su *SF* medio (observado durante la ejecución). Por lo tanto, las primeras aplicaciones de las cargas de trabajo reciben por parte del algoritmo HSP una mayor fracción de ciclos de core rápido que las restantes aplicaciones.

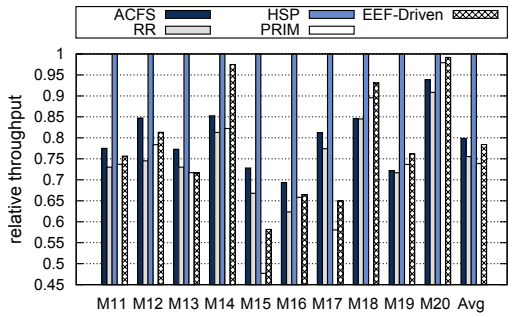
7.3.1.2. Análisis de los resultados

La figura 7.4 muestra los valores de *EDP*, *ASP* y *Unfairness* para las cargas de trabajo M1-M10 ejecutadas en la configuración 2F-4S para los distintos algoritmos de planificación. Los valores de EDP y ASP están normalizados con respecto a los resultados del algoritmo HSP, y las cargas de trabajo están ordenadas por el *EDP* relativo dado por el algoritmo EEF-Driven.

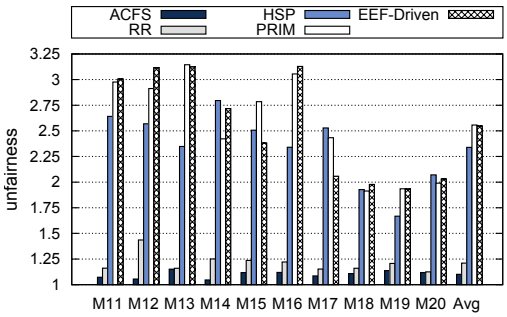
Los resultados experimentales muestran tendencias similares a las obtenidas en nuestro



(a) EDP relativo



(b) ASP relativo



(c) Unfairness

Figura 7.5: Resultados para las cargas de trabajo M11-M20 bajo la configuración 1F-3S

estudio teórico. Claramente, optimizar una métrica (por ejemplo *EDP*) puede llevar a la degradación de otra métrica (como el *ASP*). Como puede observarse, el algoritmo HSP, que intenta maximizar el rendimiento, alcanza los mejores valores de *ASP* para todas las cargas de trabajo. Por otro lado, el algoritmo EEF-Driven obtiene los mejores valores de *EDP* (cuanto más bajo mejor) para todas las cargas de trabajo. Sin embargo, HSP y EEF-Driven sufren de degradación en *EDP* y rendimiento, respectivamente. Por ejemplo, HSP puede experimentar hasta un 20 % de degradación en *EDP* con respecto a EEF-Driven a cambio de un 2.5 % de incremento en el rendimiento global (por ejemplo para la carga de trabajo M6). Por el contrario, para algunas cargas de trabajo (como por ejemplo M10) el algoritmo EEF-Driven puede obtener mejoras modestas de *EDP* con respecto a HSP (5 % en M10) a expensas de una degradación de rendimiento significativa (21 % en M10). Notablemente, la degradación del rendimiento global relativo para M10 es incluso mayor bajo la estrategia PRIM (27.5 %), que alcanza una reducción en *EDP* de sólo un 2.5 % comparado con HSP para esa misma carga de trabajo. Por otro lado, los resultados revelan que RR y ACFS, que intentan optimizar la justicia, obtienen mejoras en este aspecto a costa de degradar el*EDP* y el rendimiento global, comparado con otras estrategias que buscan optimizar el rendimiento global (HSP) o intentan mejorar la eficiencia energética (EEF-Driven y PRIM).

A continuación, analizamos los algoritmos diseñados para mejorar la eficiencia energética: PRIM y EEF-Driven (nuestra propuesta). PRIM, intenta mejorar la eficiencia energética realizando intercambios de hilos que generen ahorro de energía. Para lograr este objetivo,

los hilos con la tasa más baja de energía por instrucción sobre un core rápido (EPI_{fast}) generalmente se asignan a cores rápidos por períodos de tiempo más largos en comparación a otros hilos. Observamos que para algunas cargas de trabajo (como M1, M5 o M6) este algoritmo realiza una distribución de ciclos de cores rápidos similar a EEF-Driven, por lo tanto los resultados de *EDP* observados son similares. En estos casos, las aplicaciones con la tasa de EPI_{fast} más baja exhiben valores altos de *EEF*. Para el resto de las cargas de trabajo, EEF-Driven claramente mejora a PRIM tanto en *EDP* (una reducción en promedio de hasta el 15 % para M2) como en rendimiento global (un incremento promedio de hasta el 20 % para M2). Esta diferencia se debe a que las aplicaciones exhiben valores similares deEPI_{fast} pero valores diferentes de *EEF* (debido a tener *SF* distinto). En el caso particular donde dos hilos ingresan a una fase de programa con valores similares de EPI_{fast}, el algoritmo EEF-Driven asigna preferentemente a un core rápido el hilo con mayor *SF*. Por el contrario, como PRIM no tiene en cuenta el ratio entre *SF* y EPI_{fast} (conocida como *EEF*), realiza asignaciones de hilos a core que no optimizan *EDP*, y como consecuencia se produce una degradación significativa en el rendimiento. También observamos, que la naturaleza aleatoria de los intercambios de hilos realizados por PRIM provoca asignaciones de hilos a cores subóptimas por períodos cortos de tiempo, y esto conlleva a obtener valores de *EDP* peores de los esperados en comparación a EEF-Driven.

Con respecto a los planificadores orientados a justicia, los resultados revelan que ACFS obtiene mejor rendimiento global y justicia que RR para todas las cargas de trabajo consideradas. Esto se debe al hecho que ACFS tiene en cuenta el *SF* de las aplicaciones a la hora de tomar decisiones de planificación, a diferencia de RR. Los resultados también revelan que ambos planificadores alcanzan valores de *EDP* significativamente peores (hasta un 25 % más altos en M4) que aquellos obtenidos por EEF-Driven.

Analizamos ahora los resultados de las cargas de trabajo M11-M20, ejecutadas sobre la configuración 1F-3S. La figura 7.5 muestra los valores de *EDP* relativo, *ASP* relativo y *unfairness* para las distintas cargas de trabajo y planificadores. Los resultados revelan tendencias similares a las obtenidas con la configuración 2F-4S. Sin embargo, se pueden hacer dos observaciones importantes. En primer lugar, bajo la configuración 1F-3S, el algoritmo EEF-Driven presenta en promedio una mayor reducción de *EDP* con respecto al algoritmo HSP (20 %) a diferencia de la observada en 2F-4S (13.5 %). Asimismo, también se incrementa la degradación de rendimiento promedio con respecto a HSP (de 9 % en 2F-4S a 21,5 % en 1F-3S). Para las cargas de trabajo M11-M20, la primera aplicación de cada fila de la tabla 7.7 (por ejemplo *equake* en M11) es la única que consigue una fracción mayor de ciclos de core rápido bajo HSP. Esto contrasta con lo que hace EEF-Driven, dado que este algoritmo asigna al core rápido la segunda aplicación (por ejemplo *hmmer* en M11) durante gran parte de su ejecución, debido a que posee mayor *EEF*. Si estuviera disponible otro core rápido (como en el caso de 2F-4S), ambos algoritmos asignarían una fracción de ciclos de core rápido no despreciable para ambas aplicaciones, lo que permite obtener valores cercanos de *EDP* y *ASP* (como los obtenidos en 2F-4S). En segundo lugar, observamos que bajo la configuración 1F-3S, los algoritmos orientados a garantizar justicia (RR y ACFS) alcanzan mejores valores de *EDP* que HSP en

todos los casos; no ocurre lo mismo en la configuración 2F-4S. Intuitivamente, la fracción de ciclos de core rápido para todas las aplicaciones (incluyendo aquellas con valores altos de EPI_{fast}) se incrementa bajo RR y ACFS en la configuración 2F-4S, debido al core rápido extra. Como resultado se obtiene un mayor consumo de energía y, en consecuencia, un valor de *EDP* mayor.

7.3.2. Efectividad de ACFS-E

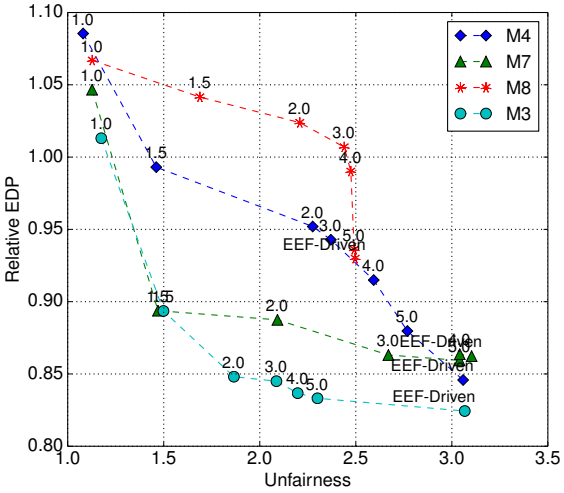
Las estrategias de planificación evaluadas en la sección anterior ofrecen un compromiso fijo entre la justicia, el rendimiento global y la eficiencia energética. Para lograr una estrategia más flexible y configurable diseñamos ACFS-E. Este planificador está equipado con dos parámetros de configuración: *EDP_factor* y *Unfairness_factor* (*UF*). Cuando estos parámetros se establecen a sus valores por defecto, el algoritmo se comporta como la implementación base del algoritmo ACFS, que intenta obtener el óptimo *unfairness* para el máximo valor de *ASP* alcanzable. Cuando el valor por defecto de alguno de estos parámetros [2] cambia, ACFS-E establece la prioridad dinámica de un hilo basándose en su *SF* (para *Unfairness_factor*) o su *EEF* (*EDP_factor*). Mediante estos parámetros de configuración el administrador del sistema puede modificar la relación entre *EDP* y justicia o entre rendimiento global y justicia.

La figura 7.6a muestra cómo el ajuste del parámetro *EDP_factor* afecta la justicia y el *EDP* para cuatro cargas de trabajo seleccionadas de la tabla 7.7 sobre la configuración 2F-4S. Los resultados revelan que el valor más bajo posible de *EDP_factor* (valor por defecto) garantiza el mejor valor de *unfairness*, mientras que valores más altos de este parámetro siempre tienden a reducir el *EDP* a expensas de degradar la justicia. Como puede observarse, al incrementar gradualmente el valor de *EDP_factor*, los valores de *EDP* y *unfairness* bajo el algoritmo ACFS-E se aproximan a aquellos alcanzados por el planificador EEF-Driven, que optimiza el *EDP*.

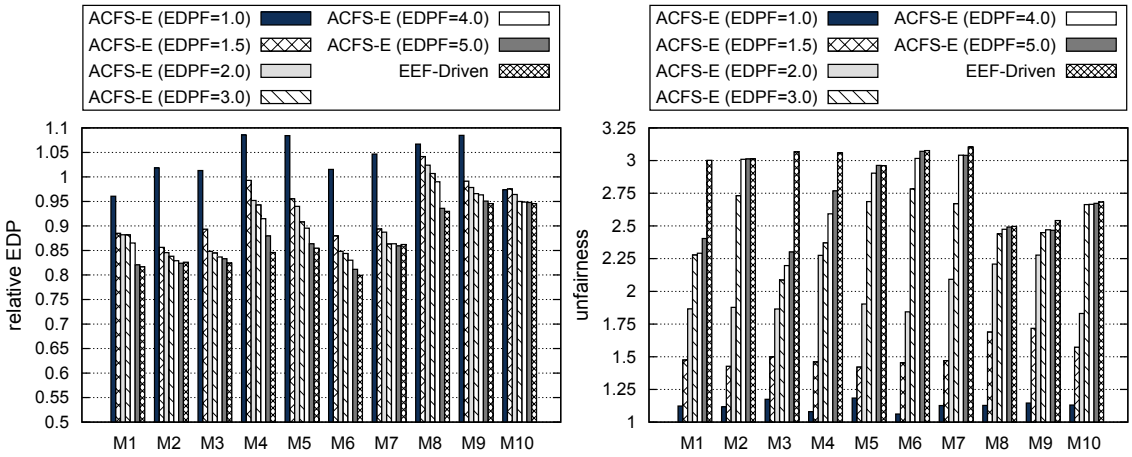
Las figuras 7.6b y 7.6c muestran la variación de EDP y *unfairness* (respectivamente) para las 10 primeras cargas de trabajo de la tabla 7.7 (M1-M10) usando valores de *EDP_factor* comprendidos entre 1 y 5. El comportamiento observado para las cargas de trabajo seleccionadas también se observa para la combinación de programas restantes: cuanto mayor sea el valor de *EDP_ factor*, menor será el EDP y mayor el *unfairness*. Los resultados también revelan que un valor de *EDP_factor* igual a 5 permite a ACFS-E alcanzar valores de EDP en un rango de 1 % relativos al planificador EEF-Driven, para todas las cargas de trabajo exploradas excepto M4. No obstante, pudimos observar que si incrementamos el valor de EDP factor por encima de 5 (*EDP_factor* aproximadamente 6,5) para la carga de trabajo M4, ACFS-E alcanza una distribución de ciclos de core rápido entre aplicaciones similar a la realizada por EEF-Driven, lo cual permite que ambos planificadores alcancen valores cercanos de *EDP* y *unfairness* (en un rango del 1 %).

Los resultados también muestran que al establecer un valor de *EDP_factor* igual a 5, ACFS-

[2] ACFS-E no permite configurar valores diferentes a 1.0 (valor por defecto) en ambos parámetros simultáneamente.



(a) Unfairness vs. EDP Relativo para las cargas de trabajo seleccionadas

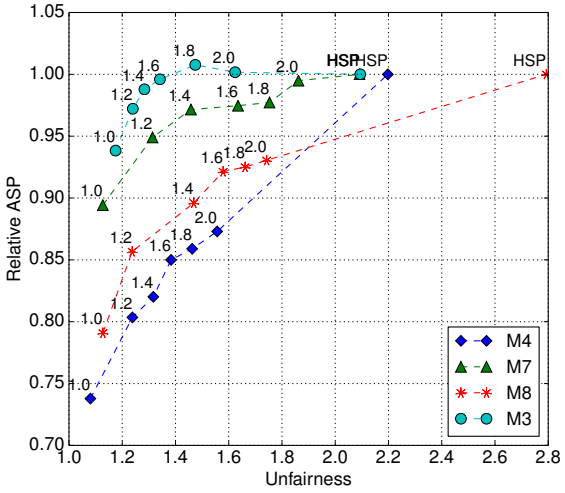


(b) EDP Relativo

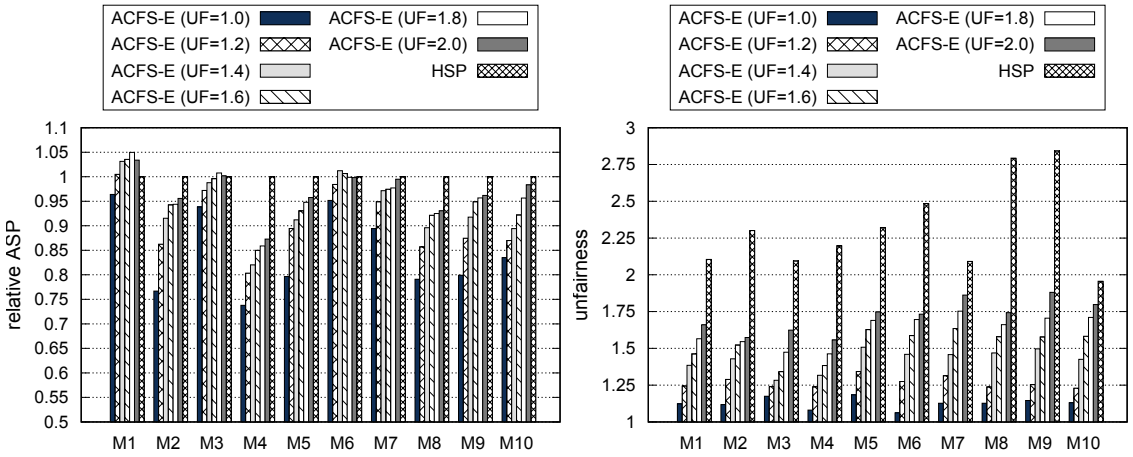
(c) Unfairness

Figura 7.6: Unfairness vs. EDP Relativo para diferentes valores de `EDP_factor` bajo el planificador ACFS-E.

E es capaz de alcanzar mejor valor de *unfairness* que EEF-Driven para las cargas de trabajo M1, M3, M4 y M9, y, al mismo tiempo obtiene resultados de eficiencia energética comparables. Este comportamiento se debe a que estas cargas de trabajo incluyen al menos tres programas que exhiben valores de *EEF* y *SF* relativamente altos durante toda la ejecución, tales como *vortex*, *galgel* o *hmmer*. En la configuración AMP utilizada, el planificador EEF-Driven asigna los dos cores rápidos disponibles a las aplicaciones de la carga de trabajo con el valor de *EEF* más alto, y al mismo tiempo relega el resto de las aplicaciones a los cores lentos. Por el contrario, en este escenario ACFS-E asigna una fracción considerable de ciclos de core rápido a todos los programas con alto *SF* y *EEF*, lo que contribuye a reducir el *unfairness* y mejora la eficiencia energética. A pesar de la reducción en el *EDP* alcanzada por ACFS-E cuando se incrementa el *EDP_factor*, esta estrategia claramente genera altos valores de *unfairness*



(a) Unfairness vs. ASP Relativo para las cargas de trabajo seleccionadas



(b) ASP Relativo

(c) Unfairness

Figura 7.7: Unfairness vs. ASP Relativo para diferentes valores de `Unfairness factor` UF bajo el planificador ACFS-E.

cuando $EDP\_factor \geq 2$.

En la sección 6.4.3 mostramos cómo en el prototipo QuickIA de Intel, el parámetro *UF* en ACFS permite ajustar el compromiso rendimiento-justicia. Los resultados en la figura 7.7a demuestran que este parámetro también es efectivo cuando ACFS-E se ejecuta sobre la placa de desarrollo ARM Juno. Más concretamente, la figura muestra el impacto sobre la justicia y el rendimiento global al variar el *UF* para las cuatro cargas de trabajo consideradas anteriormente. En general observamos que los valores altos de *UF* mejoran el rendimiento global a expensas de degradar la justicia. Además, el uso de valores más altos de este parámetro, permiten acercar el valor de ASP proporcionado por ACFS-E al del planificador HSP, intentando optimizar el rendimiento global.

Las figuras 7.7b y 7.7c muestran como varía el rendimiento global y el *unfairness* (respectivamente) para las cargas de trabajo M1-M10 de la tabla 7.7 al ajustar el valor de *UF* en un rango de 1 a 2 en pasos de 0,2. La tendencia observada para las cargas de trabajo seleccionadas también se observa para la cargas de trabajo restantes: a mayor valor de *UF*, el comportamiento de ACFS-E se aproxima al de HSP. Para la mayoría de las cargas de trabajo, un valor de *UF* igual a 2 posibilita que ACFS-E alcance valores de rendimiento en un rango del 4 % con respecto a HSP. Para unas pocas cargas de trabajo, tales como M4 o M8, es necesario incrementar el *UF* por encima de 2 para permitir a ACFS-E obtener valores cercanos a HSP en términos de rendimiento global. En particular, detectamos que para lograr que ACFS-E tenga un rendimiento en el rango del 4 % de HSP con las cargas de trabajo M4 y M8, se deben utilizar valores de *UF* de 3.6 y 3, respectivamente.

Los resultados de las figuras 7.7b y 7.7c también revelan que para la mayoría de las cargas de trabajo, los valores más altos de *UF* permiten mejorar el rendimiento global a costa de degradar la justicia. En contra de nuestras expectativas iniciales, éste no es exactamente el comportamiento observado para las cargas M1, M3 y M6. Para esta combinación de programas, ACFS-E es capaz de alcanzar mejoras en el rendimiento global cercanas al planificador HSP (hasta un 5 % de mejora en la carga M1). Sin embargo, el valor de rendimiento máximo observado no se alcanza cuando se usa el valor más alto de *UF* explorado. Por el contrario, el rendimiento global cae cuando se incrementa *UF* mas allá de cierto punto (por ejemplo: 1.8 para M1 y 1.4 para M6). Detectamos que este comportamiento se debe a los efectos de contención en recursos compartidos, que se manifiestan en las combinaciones de programas M1, M3 y M6, compuestas por aplicaciones intensivas en memoria y sensibles al hecho de compartir la cache tales como *equake*, *soplex* y *galgel*.

Para entender mejor los resultados de las cargas M1, M3 y M6, debemos resaltar que ninguno de los planificadores analizados en esta tesis (algunos propuestos por otros autores) trata el problema de los efectos de la contención por recursos compartidos. Las aplicaciones intensivas en memoria mencionadas anteriormente, presentes en estas cargas de trabajo, son las que exhiben un *speedup* más alto en la carga. Los beneficios que experimentan estas aplicaciones en los cores rápidos se deben en parte a que este tipo de core tiene una cache L2 mayor que la de los cores lentos (ver tabla 2.1 para más detalles). Desafortunadamente, cuando el planificador del SO asigna simultáneamente dos programas intensivos en memoria a los cores rápidos, las aplicaciones compiten entre ellas por el espacio en la cache L2 compartida y también por el ancho de banda del bus. Esto causa una degradación significativa en el rendimiento de ambas aplicaciones y en consecuencia se degrada también el rendimiento global.[3]

En particular, detectamos que para las cargas M1, M3 y M6, el planificador HSP asigna

[3] Observamos también que la contención en la cache L2 compartida entre cores lentos con pipeline de ejecución en orden no es tan alta, lo que provoca una degradación en rendimiento menor, y más uniforme entre las aplicaciones de las cargas de trabajo exploradas. Conjeturamos que este grado menor de contención de cache se debe al hecho de que un core con pipeline de ejecución en orden no puede gestionar multiples fallos de cache pendientes de forma simultánea; esto provoca una menor tasa de accesos al último nivel de cache por ciclo.

aplicaciones intensivas en memoria a cores rápidos simultáneamente por períodos más largos de tiempo en comparación a ACFS-E, cuando se utiliza la configuración por defecto de *UF* (1.0). A medida que incrementamos el *UF*, el planificador ACFS-E asigna una fracción mayor de ciclos de core rápido a aplicaciones con alto *speedup* (en este caso, las aplicaciones intensivas en memoria). Claramente, un leve incremento de la fracción de ciclos de core rápido a las aplicaciones intensivas en memoria con alto speedup se traduce en un mayor rendimiento para estas aplicaciones y, en consecuencia, en ganancias significativas de rendimiento global. Sin embargo, incrementar demasiado la cantidad de tiempo que estas aplicaciones se ejecutan en cores rápidos (esto ocurre como resultado de incrementar *UF* por encima de un cierto valor) conlleva a que estas aplicaciones intensivas en memoria sean asignadas simultáneamente a cores rápidos con mayor frecuencia, lo cual afecta negativamente al rendimiento de las aplicaciones individuales. Esta observación sugiere que para cargas de trabajo compuestas por múltiples aplicaciones intensivas en memoria con alto speedup, tener en cuenta los efectos de contención de recursos compartidos cuando se toman decisiones de planificación puede ayudar a mejorar aún más el rendimiento del sistema en AMPs. Conseguir que el planificador sea consiente de problemas de contención en recursos compartidos constituye una línea de investigación a futuro.

Para finalizar, vale la pena destacar que nuestro estudio demuestra que los parámetros de configuración *EDP_factor* y *UF* permiten mejorar gradualmente la eficiencia energética o el rendimiento del sistema en ACFS-E, cuando las restricciones de justicia no son tan estrictas. Además, utilizando valores altos para estos parámetros, tales como aquellos usados en nuestros experimentos, se puede configurar ACFS-E para aproximar el comportamiento de las estrategias EEF-Driven o HSP, que optimizan la eficiencia energética y el rendimiento global, respectivamente. Por lo tanto, la principal conclusión de nuestro estudio es que ACFS-E constituye una estrategia flexible y versátil, que posibilita al administrador del sistema perseguir varios objetivos de optimización con un único algoritmo de planificación.

7.4. Resumen del capítulo y conclusiones

En este capítulo extendemos el modelo teórico presentado en el capítulo 5 para mostrar la interrelación entre la justicia, la eficiencia energética y el rendimiento global en AMPs. Empleando este modelo extendido y un simulador basado en él, llevamos a cabo un estudio que nos permitió hallar el planificador teórico *EDP-Opt*, que obtiene el producto energía retardo (EDP) óptimo para el mayor valor de rendimiento global alcanzable. Nuestro estudio pone de manifiesto que la justicia, el rendimiento global y la eficiencia energética constituyen objetivos contrapuestos que no pueden ser optimizados simultáneamente. Nuestro estudio también revela que *Opt-EDP* puede aproximarse haciendo uso del *factor de eficiencia energética* (*EEF*) de una aplicación. En particular, el *EDP* puede reducirse dedicando los cores rápidos a las aplicaciones de la carga de trabajo que exhiben los valores de *EEF* más altos. Esta idea nos permitió guiar el proceso de diseño de los algoritmos EEF-Driven y ACFS-E propuestos en este capítulo.

La mayoría de los algoritmos que intentan optimizar el rendimiento o la eficiencia energética (como HSP y EEF-Driven, respectivamente) proporcionan un compromiso fijo entre rendimiento global, justicia y eficiencia energética. Para cubrir esta limitación propusimos ACFS-E, la estrategia de planificación más versátil de esta tesis doctoral. ACFS-E es una variante del algoritmo ACFS que tiene en cuenta el *EEF* de los hilos y permite al usuario configurar la relación entre la eficiencia energética y la justicia mediante el parámetro de configuración (*EDP_factor*).

Cabe destacar que los planificadores EEF-Driven y ACFS-E precisan de un mecanismo que les permita obtener el *EEF* de un hilo en tiempo de ejecución. Lamentablemente, el hardware asimétrico comercial disponible en la actualidad no permite medir el *EEF* de un hilo en tiempo de ejecución, cuando éste se ejecuta simultáneamente con otros hilos en el sistema AMP. Por esta razón, nuestra implementación de ACFS-E y EEF-Driven estiman el *EEF* online. Para lograr este objetivo desarrollamos modelos precisos de estimación de *EEF* para cada tipo de core, basados en contadores hardware. Para la construcción de todos los modelos de estimación utilizamos la estrategia *Phase-SF* propuesta en el capítulo 6.

Para evaluar la efectividad de EEF-driven, realizamos un análisis experimental donde comparamos este algoritmo con otros algoritmos para AMPs. Implementamos estos algoritmos en el kernel Linux y los evaluamos sobre la placa de desarrollo ARM Juno. Nuestros resultados experimentales revelan que el algoritmo EEF-Driven reduce el *EDP* y mejora el rendimiento sustancialmente comparado con PRIM [65].

Para concluir, mostramos la efectividad del algoritmo ACFS-E en lo que respecta a su capacidad para optimizar distintas métricas en el AMP. ACFS-E constituye el primer planificador consciente de la asimetría de la plataforma, que permite elegir la métrica a optimizar (justicia, eficiencia energética o rendimiento global) con un único algoritmo, gracias a sus dos parámetros configurables: *EDP_factor* y *Unfairness_factor*.

8 Conclusiones y trabajo futuro

Los procesadores multicore asimétricos o AMPs (*Asymmetric Multicore Processors*) con repertorio común de instrucciones han sido propuestos como alternativa a los CMPs convencionales para ofrecer un mejor rendimiento por *watt* y por unidad de área [29], así como para satisfacer las distintas demandas de cargas de trabajo diversas. En este sentido, los cores rápidos de alto rendimiento (*big o fast cores*) del AMP son adecuados para la ejecución de aplicaciones secuenciales intensivas en cómputo y para acelerar fases secuenciales de aplicaciones paralelas; y sus cores lentos de bajo consumo (*small o slow cores*) en general ofrecen mayor eficiencia energética para aplicaciones intensivas en memoria y para la ejecución de aplicaciones paralelas altamente escalables.

Con el objetivo de trasladar los beneficios de los AMPs a las aplicaciones sin requerir su modificación, el planificador del SO debe ser consciente de la asimetría de la arquitectura. Al inicio de esta tesis doctoral, la mayor parte de los algoritmos de planificación para AMPs perseguían optimizar únicamente el rendimiento global del sistema. Sin embargo, otros aspectos críticos, como la optimización de la justicia, la reducción del consumo energético, o el soporte efectivo de prioridades, no habían recibido suficiente atención por parte de la comunidad científica. El principal objetivo de esta tesis doctoral ha sido llenar este vacío mediante el diseño, implementación y evaluación de estrategias de planificación para AMPs a nivel de sistema operativo, conscientes de la justicia y de la eficiencia energética, garantizando al mismo tiempo un rendimiento global aceptable. En particular, propusimos los algoritmos de planificación Prop-SP, ACFS, EEF-Driven y ACFS-E.

Para lograr nuestros objetivos, superamos tres desafíos importantes: desarrollamos un *framework* de planificación sobre el planificador de un SO real; definimos métricas específicas para cuantificar el grado de alcance de los distintos objetivos del planificador (rendimiento global, justicia y eficiencia energética); y proporcionamos al planificador del SO de un mecanismo para estimar, en tiempo de ejecución, el *speedup* que una aplicación obtiene al usar los distintos tipos de cores en un AMP.

Nuestro *framework* de planificación nos permitió implementar y evaluar distintas estrate-

gias de planificación empleando un SO de propósito general sobre hardware asimétrico real. Debido a la complejidad que conlleva realizar modificaciones en el planificador de un SO real, muchos investigadores evaluaron sus propuestas de planificación utilizando simuladores [30, 8, 11, 65] o prototipos de planificación en espacio de usuario [45, 46]. Sin embargo, esta aproximación puede ocultar serias limitaciones de los algoritmos de planificación que se manifiestan al emplear un entorno más realista [60, 59, 56].

Para medir la efectividad de las estrategias de planificación propuestas en esta tesis, definimos nuevas métricas y adaptamos métricas existentes a entornos AMP. Esto fue necesario porque la mayor parte de las métricas previamente propuestas habían sido diseñadas para sistemas simétricos, y observamos que muchas de ellas no resultan adecuadas para multicore asimétricos. Mediante el uso de estas nuevas métricas derivamos modelos analíticos y herramientas auxiliares para aproximar la planificación teórica (reparto ideal de ciclos de cores rápidos y lentos entre aplicaciones) que optimiza distintos objetivos.

En esta tesis doctoral diseñamos un mecanismo basado en el uso de contadores *hardware* para estimar el *speedup* de una aplicación en tiempo de ejecución. Las distintas aplicaciones de una carga de trabajo multiprogramada pueden obtener un beneficio relativo o *speedup* muy distinto al ejecutarse en cores rápidos con respecto a hacerlo en cores lentos. El planificador del SO puede explotar este beneficio para tomar decisiones de planificación. Para una aplicación secuencial el *speedup* coincide con el *speedup factor* del único hilo en ejecución. Se ha demostrado que los modelos de estimación permiten al planificador obtener valores de *SF* más precisos[60, 59] con respecto a hacerlo mediante medición directa (*IPC-Sampling*) [30, 8, 64]. Para aproximar el *speedup* de una aplicación multi-hilo como un todo, derivamos fórmulas analíticas que consideran el *SF* de sus hilos, el número de hilos de la aplicación y el número de cores rápidos de la plataforma. En nuestra evaluación experimental confirmamos que asignar aplicaciones a cores en base al *speedup* es clave para mejorar el rendimiento global, la justicia o la eficiencia energética en AMPs.

Nuestra primera propuesta de planificación es el algoritmo Prop-SP, cuyo objetivo es ofrecer un buen equilibrio entre justicia y rendimiento global, y proporcionar soporte a prioridades. Este algoritmo asigna una fracción de tiempo de core rápido a una aplicación de forma proporcional al producto de su *speedup* y su prioridad. Para garantizar que una aplicación recibe una fracción específica de tiempo en los cores rápidos, Prop-SP utiliza una estrategia basada en créditos inspirada en el planificador *Credit Scheduler* de Xen [9]. Prop-SP asigna créditos de cores rápidos a las aplicaciones de forma periódica. A medida que un hilo se ejecuta sobre un core rápido, consume créditos; al agotar todos sus créditos, el planificador intentará intercambiar este hilo con otro que esté asignado a un core lento y posea créditos de cores rápidos. Además, Prop-SP ofrece soporte específico para aplicaciones multi-hilo: los créditos asignados a una aplicación paralela se distribuyen entre sus hilos activos. Para ello, Prop-SP soporta distintas estrategias de distribución de créditos que permiten satisfacer las necesidades de diversos tipos de aplicaciones. Cabe destacar que al diseñar y evaluar Prop-SP, también exploramos los beneficios que se derivan de la interacción entre el *runtime system*

y el SO. En particular, estudiamos los beneficios de la interacción *runtime system-SO* para aplicaciones OpenMP regulares. De nuestra evaluación experimental podemos concluir lo siguiente:

- Prop-SP ofrece mejor compromiso rendimiento-justicia que otras estrategias de planificación propuestas previamente, como A-DWRR[33], el algoritmo de referencia hasta la fecha.
- La clave del éxito de Prop-SP reside en que tiene en cuenta el *speedup* de las aplicaciones, ofrece soporte para aplicaciones multi-hilo y brinda mejor soporte a prioridades de usuario que otros algoritmos.

Si bien Prop-SP consigue una mejora sustancial en la justicia frente a otras propuestas, hasta el momento desconocíamos si era capaz de optimizar la justicia en un AMP. Para proporcionar una respuesta a esta cuestión, desarrollamos un modelo teórico para aproximar de forma analítica el rendimiento global y el grado de justicia alcanzados por una estrategia de planificación al utilizar cargas de trabajo multiprogramadas en AMPs. Empleando este modelo teórico y un simulador basado en él, realizamos un estudio para aproximar el planificador teórico óptimo de justicia, es decir, la estrategia de reparto de ciclos de cores rápidos y lentos entre aplicaciones que asegura el máximo rendimiento alcanzable para el valor óptimo de justicia. A partir de los resultados de nuestro análisis teórico concluimos que:

- Prop-SP consigue un buen compromiso entre rendimiento y justicia, pero no es capaz de alcanzar el comportamiento del planificador óptimo de justicia.
- En la mayoría de los casos no es posible optimizar la justicia y el rendimiento simultáneamente en un AMPs. Es decir, la optimización de la justicia y del rendimiento en multicore asimétricos constituyen objetivos claramente contrapuestos.

A partir de los resultados del análisis teórico diseñamos ACFS, un algoritmo que aproxima el comportamiento del planificador teórico que optimiza la justicia en un AMP. ACFS equilibra la degradación en rendimiento que experimentan las diversas aplicaciones de una carga de trabajo como resultado de compartir el sistema asimétrico. Para realizar un seguimiento de la degradación en rendimiento acumulada por una aplicación a lo largo del tiempo, cada hilo tiene asociado un contador de progreso, que se actualiza en base al *speedup* de la aplicación y al tipo de core usado para ejecutar cada fase de ejecución. ACFS también da soporte a prioridades de usuario y aplicaciones multi-hilo. Además, permite ajustar gradualmente el nivel relativo de justicia y rendimiento global mediante un parámetro de configuración del planificador denominado *unfairness factor* o *UF*.

Previo al diseño e implementación de ACFS, tuvimos acceso al prototipo de multicore asimétrico QuickIA de Intel, un sistema dual socket compuesto por un procesador Intel

Xeon E5450 y un procesador Intel Atom N330. Este prototipo nos permitió identificar ciertos desafíos que no se habían manifestado previamente en entornos asimétricos emulados o infraestructuras de simulación. Uno de los desafíos más significativos al que tuvimos que hacer frente durante la tesis doctoral fue diseñar un mecanismo para estimar el *SF* de un hilo en tiempo de ejecución para este sistema. Específicamente, vimos que las metodologías utilizadas anteriormente para construir modelos de estimación de *SF* no resultaban efectivas en el prototipo QuickIA. Construimos una nueva metodología que utiliza información asociada a las diferentes fases de ejecución de los programas, y a partir de esta información, derivamos un modelo preciso para predecir el *SF* en el QuickIA. La nueva metodología constituye una contribución clave en esta tesis doctoral y nos permitió construir modelos de estimación precisos de *SF* sobre otras plataformas asimétricas reales.

Las conclusiones obtenidas de nuestra evaluación experimental del planificador ACFS son las siguientes:

- ACFS obtiene mejores valores de justicia y rendimiento en comparación con los algoritmos A-DWRR [33], EQP[64] y que nuestra propuesta inicial Prop-SP, para cargas de trabajo que incluyen tanto aplicaciones secuenciales como multi-hilo.
- Las mejoras alcanzadas por ACFS en el escenario de aplicaciones multi-hilo se deben principalmente a que este algoritmo tiene en cuenta el *speedup* de las aplicación como un todo, a diferencia de EQP[1] [64] que considera el *SF* de los hilos individualmente.
- ACFS obtiene mejoras significativas en cuanto a justicia cuando se tienen en cuenta prioridades de usuario, con respecto a A-DWRR y Prop-SP.

Durante la última etapa de la tesis doctoral exploramos estrategias de planificación orientadas a optimizar la eficiencia energética. Para guiar el proceso de diseño de estas estrategias, extendimos el modelo teórico presentado previamente con la capacidad de aproximar el producto de energía-retardo o *EDP*. Utilizando el nuevo modelo, realizamos un análisis teórico que nos permitió aproximar el planificador teórico que optimiza el *EDP*, es decir, el algoritmo que asegura el máximo rendimiento alcanzable para el valor óptimo (mínimo) de *EDP*. A continuación resumimos los aspectos más relevantes que surgieron de nuestro análisis teórico:

- A partir de los resultados obtenidos modelamos el *factor de eficiencia energética* o *EEF* de un hilo, una métrica que el planificador del SO puede explotar para mejorar la eficiencia energética que se extrae de un AMP. El EEF se define como el cociente entre el *speedup factor* de un hilo de ejecución y su ratio de energía por instrucción (*EPI*) observado en el core rápido.

[1] Esta estrategia de planificación fue propuesta por Van. Craenest y otros unos meses después de la publicación de nuestro primer trabajo sobre Prop-SP.

- En general, concluimos que el rendimiento global, la justicia y la eficiencia energética son objetivos contrapuestos, que no pueden ser optimizados simultáneamente en un AMP.

El análisis anterior fue clave para guiar el proceso de diseño de nuestras propuestas de planificación EEF-Driven y ACFS-E. Estos algoritmos utilizan el *EEF* de cada hilo para realizar asignaciones de hilos a cores y de esta forma reducir el *EDP*. Tanto EEF-Driven como ACFS-E precisan de un mecanismo para obtener el *EEF* de un hilo en tiempo de ejecución. Para evaluar la efectividad de estos algoritmos, empleamos la placa de desarrollo ARM Juno, un sistema asimétrico comercial que permite al usuario tomar medidas de consumo energético, una característica que no poseen los sistemas asimétricos utilizados durante etapas previas de la tesis doctoral. A pesar de esta característica del sistema, existen aún limitaciones inherentes que impiden medir el *EEF* de un hilo individual, cuando éste se ejecuta con otros hilos en el sistema. Para superar esta barrera, adaptamos nuestra metodología diseñada previamente para obtener modelos de estimación de *SF*, y la aplicamos sobre la placa de desarrollo ARM Juno, lo que nos permitió construir modelos de estimación precisos de *EEF* para esta plataforma.

El objetivo del algoritmo EEF-Driven es aproximar el planificador teórico que optimiza el *EDP*. Específicamente, EEF-Driven asigna a los cores rápidos aquellas aplicaciones de la carga de trabajo con el valor más alto de *EEF*. A partir de los resultados de nuestro análisis experimental podemos concluir que EEF-Driven reduce el *EDP* y mejora el rendimiento significativamente comparado con otros algoritmos del estado del arte (PRIM[65] y RR[8]).

La mayoría de los algoritmos de planificación que intentan optimizar sólo un aspecto (rendimiento global, justicia o eficiencia energética) mantienen un compromiso fijo entre estas tres métricas. Por el contrario, nuestro planificador ACFS-E, variante de ACFS, permite al usuario ajustar el compromiso rendimiento-justicia o energía-justicia, mediante el incremento gradual del rendimiento global o de la eficiencia energética en escenarios donde los requerimientos de justicia no son tan estrictos. Asimismo, ACFS-E constituye la primera estrategia de planificación para AMPs que puede configurarse para optimizar cualquiera de estos tres aspectos individualmente, con un único algoritmo de planificación. Para lograr estos objetivos, este planificador está provisto de dos parámetros de configuración, UF (parámetro de ACFS) y EDP_factor, que el SO emplea internamente para ajustar dinámicamente la prioridad de los hilos en base a su *speedup* o a su *EEF*. ACFS-E es sin duda la propuesta de planificación más versátil de esta tesis doctoral.

Para finalizar, podemos concluir que la investigación realizada en esta tesis doctoral ha permitido llenar un importante vacío en cuanto a la optimización de la justicia y la eficiencia energética, así como en la mejora del soporte a prioridades en AMPs.

8.1. Líneas de trabajo futuro

A pesar de los avances realizados en esta tesis en cuanto a planificación, aún quedan numerosos desafíos a los que hacer frente en el contexto de los procesadores multicore asimétricos. Estos desafíos dan lugar a líneas de trabajo futuro como las que se enumeran a continuación:

- *Explorar el impacto en la justicia, rendimiento y eficiencia energética que tiene la contención por recursos compartidos en AMPs.* En los sistemas multicore asimétricos actuales, como aquellos provistos de procesadores big.LITTLE de ARM, los cores del mismo tipo (organizados en *clusters*) comparten el último nivel de cache y además compiten por el ancho de banda del bus. Además todos los cores del sistema compiten por el acceso a memoria (controlador DRAM). La competencia por recursos compartidos puede afectar negativamente y de forma desigual al rendimiento de las aplicaciones de la carga de trabajo. El análisis de los beneficios de combinar decisiones a nivel de contención de recursos compartidos con decisiones de planificación asimétrica en el SO constituye una vía interesante para trabajo futuro.

- *Ofrecer soporte específico para un rango más amplio de aplicaciones.* En los análisis experimentales realizados en esta tesis doctoral se han empleado cargas de trabajo formadas por aplicaciones de diversas suites de benchmarks para HPC. Sin embargo, los AMPs también podrían ofrecer beneficios significativos a otros tipos de programas, como aplicaciones sensibles a la latencia, los servidores web, o las aplicaciones cliente típicas de las plataformas móviles.

- *Estudiar el impacto de los algoritmos de planificación sobre multicore asimétricos implementados a nivel de hipervisor o VMM (Virtual Machine Monitor)* . En la actualidad la virtualización es ampliamente utilizada y es la base de los servicios de computación sobre *Cloud*.

- *Explorar soluciones que combinen algoritmos de planificación conscientes de la asimetría en la plataforma con políticas de ajuste dinámico de la frecuencia (DVFS) para mejorar la eficiencia energética.* En esta tesis asumimos que los cores están a una frecuencia fija. En algunos procesadores es posible ajustar la frecuencia dinámicamente mediante DVFS, una técnica utilizada para gestionar el manejo de energía en una CPU. Cuando una aplicación se ejecuta en un core que opera a una frecuencia inferior a la máxima soportada, se reduce el consumo de potencia del sistema, lo que en ocasiones permite también reducir el consumo energético. Sin embargo, DVFS debe ser utilizado con cautela debido a que suele tener efectos negativos en el rendimiento de la aplicación. Es en estos casos donde la combinación de DVFS con algoritmos de planificación conscientes de la asimetría en la plataforma pueden contribuir a mejorar el compromiso entre rendimiento y eficiencia energética.

- *Diseñar mecanismos de interacción más sofisticados entre el runtime system y el SO*

para multicore asimétricos. El soporte propuesto en esta tesis doctoral (AID) sólo es apto para aplicaciones paralelas regulares y balanceadas. Sin embargo, no todas las aplicaciones multihilo presentan estas características. En aplicaciones con paralelismo a nivel de tarea, la interacción del *runtime* con el SO debería ser muy diferente a la aproximación propuesta en esta tesis doctoral. Consideramos que la exploración de distintos mecanismos de interacción entre ambos componentes software para diversos tipos de aplicaciones paralelas constituye una prometedora línea de trabajo futuro.

A Aproximación del speedup de aplicaciones

En este apéndice presentamos el proceso de derivación de las fórmulas para estimar el *speedup* de una aplicación multihilo, cuando se ejecuta sola en un sistema AMP. Nótese que el speedup en este contexto se define como el beneficio proporcionado por el planificador con respecto a una ejecución hipotética de la aplicación paralela donde sólo se usarán cores lentos. En este apéndice derivamos analíticamente una fórmula para aproximar el *speedup* bajo cada una de las tres estrategias de distribución de créditos: *Even*, *AID* y *BusyFC*.

A.1. Speedup de aplicaciones multihilo bajo Even y AID

Para la derivación del speedup en este escenario hacemos las siguientes suposiciones:

- Bajo una estrategia de distribución *Even* asumimos que la aplicación está perfectamente balanceada, es decir el trabajo se distribuye de manera uniforme entre los hilos.
- Bajo una estrategia de distribución *AID* asumimos que el trabajo se distribuye asimétricamente entre los hilos asignados a cores rápidos y a cores lentos tal como se describe en la sección 4.2.
- La aplicación consta de k fases paralelas ($k > 1$) separadas por barreras de sincronización. Debido a que la aplicación esta perfectamente balanceada, suponemos que los hilos llegan a las barreras de sincronización al mismo tiempo.
- En la mayoría de aplicaciones paralelas que examinamos, todos sus hilos tienen *speedup factors* muy similares ya que ejecutan el mismo código con diferentes datos. Por esta razón asumimos el mismo SF entre los hilos.
- Suponemos además que el número de hilos en la aplicación no excede el número de cores en el AMP.

- Al derivar las fórmulas, no se tienen en cuenta el *overhead* asociado a las migraciones de hilos.

Además de estos supuestos, introducimos la siguiente notación auxiliar:

- N: número de hilos en la aplicación.
- $CT_{SC,i}$: tiempo de ejecución de la fase paralela i-ésima que resultaría al ejecutar todos los hilos de la aplicación en cores lentos. Como la aplicación multihilo está perfectamente balanceada, $CT_{SC,i}$ se corresponde con el tiempo de ejecución de cualquiera de sus hilos para esa fase paralela. Nótese que $CT_{slow} = \sum_{i=1}^{k} CT_{SC,i}$.
- NI_i: número de instrucciones que la aplicación como un todo ejecuta en la i-ésima fase paralela.

A.1.1. Speedup bajo Even

Para derivar el *speedup* bajo esta estrategia de distribución de créditos introducimos la siguiente notación:

- N_{FC}, N_{SC} número de cores rápidos y lentos, respectivamente.
- CT_{even}: tiempo de ejecución de una aplicación bajo Even.
- $CT_{even,i}$: tiempo de ejecución de la i-ésima fase paralela de la aplicación bajo Even, tal que $CT_{even} = \sum_{i=1}^{k} CT_{even,i}$. Dado que Even asigna la misma fracción de tiempo de core rápido a todos los hilos y la aplicación está perfectamente balanceada,$CT_{even,i}$ se corresponde con el tiempo de ejecución de cualquiera de sus hilos para la fase paralela.
- $NI_{T,i}$: Número de instrucciones ejecutadas por un hilo de la aplicación en la i-ésima fase paralela. Como la aplicación está balanceada, todos los hilos ejecutan el mismo número de instrucciones hasta alcanzar la barrera. Por lo tanto, $NI_{T,i} = \frac{NI_i}{N}$
- $F_{fast,i}, F_{slow,i}$: fracción de tiempo $CT_{even,i}$ que los hilos ejecutan en cores rápidos y lentos respectivamente en la i-ésima fase paralela. Bajo la estrategia Even, los hilos reciben la misma fracción de tiempo de core rápido $F_{fast,i} = \frac{N_{FC}}{N}$. Para asegurar esta distribución ideal de ciclos de core rápido, los hilos deben intercambiarse entre los cores rápidos y lentos en un período de tiempo infinitamente pequeño durante la fase paralela.
- $f_{fast,i}, f_{slow,i}$: fracción de instrucciones de $NI_{T,i}$ que los hilos ejecutan sobre cores rápidos y lentos respectivamente durante la i-ésima fase paralela. Nótese que $f_{fast,i} + f_{slow,i} = 1$

Podemos aproximar CT_{slow} con el tiempo que cualquier hilo de la aplicación tarda en ejecutar todas sus instrucciones sobre un core lento:

$$\begin{aligned} CT_{slow} &= \sum_{i=1}^{k} CT_{SC,i} \\ &= \sum_{i=1}^{k} SPI_{SC} \cdot NI_{T,i} \\ &= \sum_{i=1}^{k} SPI_{SC} \cdot \frac{NI_i}{N} \\ &= \sum_{i=1}^{k} SF \cdot SPI_{FC} \cdot \frac{NI_i}{N} \end{aligned} \tag{A.1}$$

De la misma forma, CT_{even} se puede expresar como sigue:

$$CT_{even} = \sum_{i=1}^{k} CT_{even,i} \tag{A.2}$$

Para cada fase paralela $i \epsilon (1 \dots k)$ obtenemos lo siguiente:

$$\begin{aligned} CT_{even,i} &= f_{fast,i} \cdot NI_{T,i} \cdot SPI_{FC} + f_{slow,i} \cdot NI_{T,i} \cdot SPI_{SC} \\ &= NI_{T,i}(f_{fast,i} \cdot SPI_{FC} + f_{slow,i} \cdot SPI_{SC}) \\ &= NI_{T,i}(f_{fast,i} \cdot SPI_{FC} + (1 - f_{fast,i}) \cdot SPI_{SC}) \\ &= NI_{T,i}(f_{fast,i} \cdot SPI_{FC} + (1 - f_{fast,i}) \cdot SF \cdot SPI_{FC}) \\ &= \frac{NI_i}{N} \cdot SPI_{FC} \cdot (f_{fast,i} + (1 - f_{fast,i}) \cdot SF) \end{aligned} \tag{A.3}$$

Por otra parte, tenemos lo siguiente:

$$\begin{aligned} CT_{even,i} \cdot F_{fast,i} &= f_{fast,i} \cdot NI_{T,i} \cdot SPI_{FC} \\ &= f_{fast,i} \cdot \frac{NI_i}{N} \cdot SPI_{FC} \end{aligned} \tag{A.4}$$

Claramente, si $N \leq N_{FC}$, Prop-SP asigna todos los hilos de la aplicación a cores rápidos ($F_{fast,i} = f_{fast,i} = 1$). Por lo tanto, el *speedup* bajo *Even* coincide con el *SF* de los hilos de la aplicación:

$$\begin{aligned} Speedup_{even(N \leq N_{FC})} &= \frac{CT_{slow}}{CT_{even}} = \frac{\sum_{i=1}^{k} CT_{SC,i}}{\sum_{i=1}^{k} CT_{even,i}} \\ &= \frac{\sum_{i=1}^{k} (SF \cdot SPI_{FC} \cdot \frac{NI_i}{N})}{\sum_{i=1}^{k} (\frac{NI_i}{N} \cdot SPI_{FC})} \end{aligned} \tag{A.5}$$

Para aproximar el *speedup* cuando $N > N_{FC}$, procedemos a derivar una ecuación que captura la relación entre $f_{fast,i}$ y $F_{fast,i}$, considerando que $F_{fast,i} = \frac{N_{FC}}{N}$ bajo *Even*. De las ecuaciones A.3 y A.4, podemos despejar $f_{fast,i}$ de la siguiente forma:

$$\begin{aligned} \frac{f_{fast,i} \cdot \frac{NI_i}{N} \cdot SPI_{FC}}{F_{fast,i}} &= \frac{NI_i}{N} \cdot SPI_{FC} \cdot (f_{fast,i} + (1 - f_{fast,i}) \cdot SF) \\ \Rightarrow f_{fast,i} &= \frac{SF}{\frac{1}{F_{fast,i}} - 1 + SF} = \frac{1}{\frac{1}{SF} \cdot (\frac{1}{F_{fast,i}} - 1) + 1} \\ \Rightarrow f_{fast,i} &= \frac{1}{\frac{1}{SF} \cdot (\frac{N}{N_{FC}} - 1) + 1} \end{aligned} \tag{A.6}$$

Combinando las ecuaciones A.1-A.4, derivamos el *speedup* para la aplicación bajo *Even* cuando $N > N_{FC}$:

$$
\begin{aligned}
Speedup_{even(N>N_{FC})} &= \frac{CT_{slow}}{CT_{even}} = \frac{\sum_{i=1}^{k} CT_{SC,i}}{\sum_{i=1}^{k} CT_{even,i}} \\
&= \frac{\sum_{i=1}^{k} (SF \cdot SPI_{FC} \cdot \frac{NI_i}{N})}{\sum_{i=1}^{k} (\frac{NI_i}{N} \cdot SPI_{FC} \cdot (f_{fast,i} + (1 - f_{fast,i}) \cdot SF))} \\
&= \frac{\sum_{i=1}^{k} (SF \cdot SPI_{FC} \cdot \frac{NI_i}{N})}{\sum_{i=1}^{k} (\frac{NI_i}{N} \cdot SPI_{FC} \cdot (f_{fast,i} \cdot (1 - SF) + SF))} \\
&= \frac{SF \cdot \frac{SPI_{FC}}{N} \cdot \sum_{i=1}^{k} (NI_i)}{\frac{SPI_{FC}}{N} \cdot (\frac{1}{\frac{1}{SF} \cdot (\frac{N}{N_{FC}} - 1) + 1} \cdot (1 - SF) + SF) \cdot \sum_{i=1}^{k} (NI_i)} \\
&= \frac{SF}{(\frac{1}{\frac{1}{SF} \cdot (\frac{N}{N_{FC}} - 1) + 1} \cdot (1 - SF) + SF)} \\
&\Rightarrow Speedup_{even(N>N_{FC})} = \frac{N_{FC}}{N} \cdot (SF - 1) + 1
\end{aligned}
\tag{A.7}
$$

En general, podemos escribir la formula de *speedup* de la siguiente manera:

$$
Speedup_{even} = \frac{MIN(N_{FC}, N)}{N} \cdot (SF - 1) + 1 \tag{A.8}
$$

A.1.2. Speedup bajo AID

Para derivar una expresión para el *speedup* bajo la estrategia de distribución *AID* introducimos la siguiente notación:

- CT_{AID}: tiempo de ejecución de la aplicación bajo *AID*.
- $CT_{AID,i}$: tiempo de ejecución de la fase paralela i-ésima de la aplicación bajo *AID* tal que $CT_{AID} = \sum_{i=1}^{k} CT_{AID,i}$. Nótese que $CT_{AID,i}$ se corresponde con el tiempo de ejecución de cualquiera los hilos para la fase paralela, ya que la distribución asimétrica de iteraciones ideal hace que los hilos lleguen a la barrera de sincronización al mismo tiempo.

- $NI_{fast,i}, NI_{slow,i}$: número de instrucciones ejecutadas en la i-ésima fase paralela por un hilo en un core rápido y un core lento, respectivamente.

- $N_{FC\_threads}, N_{SC\_threads}$: número de hilos de la aplicación asignados a cores rápidos y cores lentos, respectivamente. Obviamente, $N = N_{FC\_threads} + N_{SC\_threads}$

Para cada fase paralela $i\epsilon(1 \dots k)$ tenemos:

$$NI_i = N_{FC\_threads} \cdot NI_{fast,i} + N_{SC\_threads} \cdot NI_{slow,i} \tag{A.9}$$

Al igual que en el escenario considerado bajo la estrategia *Even*, $Speedup_{AID} = SF$ cuando $N \leq N_{FC}$. Cuando $N > N_{FC}$, Prop-SP debe asignar los cores rápidos a hilos de esta aplicación ya que ésta se ejecuta sola en el sistema. Por lo tanto, $NFC\_threads = N_{FC}$. Como todos los hilos alcanzan la barrera de sincronización al mismo tiempo podemos expresar $CT_{AID,i}$ de la siguiente manera:

$$CT_{AID,i} = NI_{fast,i} \cdot SPI_{FC} = NI_{slow,i} \cdot SPI_{SC} \tag{A.10}$$

O de forma similar:

$$NI_{fast,i} = NI_{slow,i} \cdot SF \tag{A.11}$$

Combinando las ecuaciones A.9 y A.11 obtenemos la siguiente ecuación:

$$\begin{aligned} NI_{fast,i} &= \frac{SF \cdot NI_i}{SF \cdot N_{FC\_threads} + N_{SC\_threads}} \\ &= \frac{SF \cdot NI_i}{SF \cdot N_{FC\_threads} + N - N_{FC\_threads}} \\ &= \frac{SF \cdot NI_i}{N_{FC\_threads} \cdot (SF-1) + N} \end{aligned} \tag{A.12}$$

Utilizando las ecuaciones A.1, A.10 y A.12, el *speedup* para una aplicación multihilo bajo la estrategia de distribución de créditos AID, cuando $N > N_{FC}$, puede aproximarse de la siguiente

forma:

$$
\begin{aligned}
Speedup_{AID(N>N_{FC})} = \frac{CT_{slow}}{CT_{AID}} &= \frac{\sum_{i=1}^{k}(SF \cdot SPI_{FC} \cdot \frac{NI_i}{N})}{\sum_{i=1}^{k}(NI_{fast,i} \cdot SPI_{FC})} \\
&= \frac{\sum_{i=1}^{k}(SF \cdot SPI_{FC} \cdot \frac{NI_i}{N})}{\sum_{i=1}^{k}((\frac{SF \cdot NI_i}{N_{FC\_threads} \cdot (SF-1)+N}) \cdot SPI_{FC})} \\
&= \frac{\frac{SF \cdot SPI_{FC}}{N} \cdot \sum_{i=1}^{k} NI_i}{\frac{SF \cdot SPI_{FC}}{N_{FC\_threads} \cdot (SF-1)+N}) \cdot \sum_{i=1}^{k} NI_i} \\
&= \frac{N_{FC\_threads} \cdot (SF-1) + N}{N} \\
&= \frac{N_{FC\_threads}}{N} \cdot (SF-1) + 1 \\
&= \frac{N_{FC}}{N} \cdot (SF-1) + 1
\end{aligned}
\tag{A.13}
$$

Finalmente, la fórmula general para el *speedup* puede expresarse como sigue:

$$
Speedup_{AID} = \frac{MIN(N_{FC}, N)}{N} \cdot (SF-1) + 1 \tag{A.14}
$$

A pesar de las diferencias entre *Even* y *AID*, llegamos a la misma ecuación de *speedup* para ambas estrategias de distribución de créditos. Esto se debe a que no consideramos el *overhead* de las migraciones en nuestras derivaciones teóricas. En la práctica, *AID* debería proporcionar mejor rendimiento que *Even* para la mayoría de los casos, debido al menor número de migraciones que desencadena. En particular, para conseguir la distribución ideal del tiempo de cores rápidos bajo *Even*, los hilos deben intercambiarse entre cores rápidos y lentos a una frecuencia muy elevada. Hemos observado que con el rango de frecuencias de intercambio de hilos utilizado en nuestro entorno experimental, las aplicaciones paralelas con paralelismo de grano grueso y grano medio logran un *speedup* algo más cercano al ideal que las que presentan paralelismo de grano fino.

A.2. Speedup de aplicaciones multihilo ejecutadas bajo BusyFC

Para derivar el *speedup* bajo la estrategia de distribución de créditos *BusyFC* hacemos las siguientes suposiciones:

- El número de hilos de la aplicación paralela no supera al número total de coresdel AMP. Esta suposición es razonable dado que las aplicaciones HPC intensivas en CPU no suelen ejecutarse con más hilos que cores en el sistema.
- Como mencionamos en los supuestos bajo las estrategias *Even* y *AID*, debido a que los hilos de las aplicaciones examinadas ejecutan el mismo código sobre distintos datos, el *speedup factor* de estos hilos es similar. En este caso optamos por utilizar el *speedup factor* medio de todos los hilos de la aplicación (SF_{avg}) para aproximar el SF de cualquier hilo.
- Al igual que las estrategias *Even* y *AID* no consideramos el *overhead* asociado a las migraciones de hilos al derivar la expresión para aproximar el *speedup*.
- La aplicación está perfectamente balanceada, es decir todos los hilos realizan la misma cantidad de trabajo en paralelo hasta su finalización.

También suponemos que la aplicación consta de k fases paralelas balanceadas ($k \geq 1$) ejecutadas una detrás de otra y separadas por barreras de sincronización. Dentro de una fase paralela P, todos los hilos completan el mismo número de instrucciones (NI_P) en paralelo. Nótese, que el número de instrucciones a completar en cada fase paralela puede ser diferente. Todos los hilos ejecutarán el mismo número de instrucciones hasta su terminación. Más concretamente, si NI indica el número total de instrucciones ejecutadas por cada hilo entonces $NI = \sum_{i=1}^{k} NI_i$.

El tiempo de ejecución de una aplicación paralela está determinado por el hilo más lento. Por lo tanto, para determinar el tiempo de ejecución de la aplicación es necesario el tiempo de ejecución del hilo más lento para cada fase paralela.

En una ejecución donde sólo se utilizan cores lentos, todos los hilos tardarán aproximadamente el mismo tiempo para completar las instrucciones asociadas a una fase paralela dada. Como la aplicación está perfectamente balanceada cualquier hilo puede ser considerado como el más lento. Como resultado, aproximamos este tiempo de ejecución (CT_{slow}) como el tiempo en que cualquier hilo con *speedup factor* SF_{avg} tarda en completar las NI instrucciones en el core lento.

Para determinar el tiempo de ejecución del hilo más lento en una fase paralela bajo *BusyFCs* introducimos la siguiente notación:

- NI_{FC}, NI_{SC} el número total de instrucciones que el hilo más lento completa en el core rápido y lento, respectivamente. Para una aplicación i $NI_i = NI_{FC,i} + N_{SC,i}$

- SPI_{fc}, SPI_{sc} segundos por instrucción promedio sobre core rápido y lento, respectivamente.

Partimos de $CT_{BusyFCs}$:

$$\begin{aligned} CT_{BusyFCs} &= NI_{FC} \cdot SPI_{fc} + NI_{SC} \cdot SPI_{sc} \\ &= \sum_{i=1}^{k} (NI_{FC,i} \cdot SPI_{fc,i} + NI_{SC,i} \cdot SPI_{sc,i}) \end{aligned} \tag{A.15}$$

La fórmula para el tiempo de ejecución en el core lento es:

$$\begin{aligned} CT_{slow} &= NI \cdot SPI_{sc} \\ &= \sum_{i=1}^{k} (NI \cdot SPI_{sc,i}) \end{aligned} \tag{A.16}$$

El *speedup factor* promedio se calcula como sigue:

$$SF_{avg} = \frac{SPI_{sc}}{SPI_{fc}} \tag{A.17}$$

El tiempo de ejecución del hilo más lento en una fase paralela bajo *BusyFCs*, puede obtenerse a partir de la fracción de instrucciones ejecutadas por este hilo en los cores rápidos y los cores lentos. Antes de realizar la derivación de la expresión que aproxima este tiempo de ejecución, describimos el conjunto de acciones realizadas por el algoritmo de planificación bajo $BusyFCs$ durante la ejecución de una fase paralela P dada, donde cada hilo tiene que ejecutar NI_P instrucciones para alcanzar la barrera de sincronización al final de la fase.

Algoritmo A.1: Ejecución de una fase paralela bajo BusyFCs

{ • *R es el conjunto de hilos en ejecución de una aplicación.* }

mientras $R \neq \theta$ **hacer**
  $f \leftarrow min(N_{FC}, |R|)$
  Eliminar f hilos de R y migrarlos a cores rápidos.
  Garantizar que los hilos en R se ejecuten en los cores lentos.
  Esperar a que los hilos en cores rápidos terminen (los cores rápidos quedan libres).
fin

Al principio de la fase paralela, el planificador asigna a los cores rápidos tantos hilos como sea posible (a lo sumo N_{FC} hilos), mientras que los hilos restantes se asignan a cores lentos. Los hilos en cores rápidos completan todas sus instrucciones asociadas a esta fase paralela,

Tabla A.1: Fracción del número de instrucciones ejecutadas sobre cores rápidos y lentos cuando los hilos de una aplicación paralela se ejecutan bajo BusyFC.

Iteración	$F_{fc}(i)$	$F_{sc}(i)$
1	1	$\frac{1}{SF_{avg}}$
2	$1-\frac{1}{SF_{avg}}$	$\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}$
3	$1-\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}$	$\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}}{SF_{avg}}$
⋯	⋯	⋯
i	$1-F_{sc}(i-1)$	$1-F_{sc}(i-1)+\frac{F_{fc}(i)}{SF_{avg}}$

NI_P , y luego se bloquean. Como resultado los cores rápidos quedan libres. Por el contrario, los hilos sobre cores lentos ejecutan $\frac{NI_P}{SF_{avg}}$ instrucciones durante ese período de tiempo. Tan pronto como los cores rápidos quedan libres, el planificador migra a estos cores los siguientes N_{FC} hilos. A medida que estos hilos completan sus instrucciones restantes, los cores rápidos vuelven a quedar libres. Esta secuencia de acciones se resume en el algoritmo A.1. Al emplear la estrategia BusyFCs, el planificador asigna tantos hilos como sea posible a los cores rápidos y espera hasta que estos hilos terminen. Por cada iteración se migran N_{FC} hilos a cores rápidos, por lo tanto habrá $\frac{N-1}{N_{FC}}+1$.

La tabla A.1 muestra F_{fc} y F_{sc} , la fracción de instrucciones ejecutadas en cores rápidos y lentos, respectivamente. Cada fila de la tabla muestra los valores para una iteración específica, $F_{fc}(i)$ y $F_{sc}(i)$. En la primera iteración, los hilos que se ejecutan en cores rápidos completan todas las instrucciones NI_P, mientras que los hilos sobre cores lentos completan tantas instrucciones como $\frac{NI_p}{SF_{avg}}$. De manera equivalente, estos valores pueden ser expresados en términos de una fracción de NI_P, como 1 (100 %) y $\frac{1}{SF_{avg}}$, respectivamente. En la segunda iteración, los hilos que se ejecutan en cores rápidos tiene que completar una fracción $1-\frac{1}{SF_{avg}}$. Esto se debe a que esos hilos ya completaron $\frac{1}{SF}$ en cores lentos en la iteración anterior. Por lo tanto, se ejecutarán en los cores rápidos hasta completar su fracción restante: $1-\frac{1}{SF}$.

De acuerdo con la información que se muestra en la tabla A.1, la fracción de instrucciones ejecutada por hilos asignados a cores rápidos y lentos en la iteración i, que se denota como $F_{fc}(i)$ y $F_{sc}(i)$, puede definirse recursivamente como:

$$F_{fc}(1) = 1 \tag{A.18}$$

$$F_{sc}(1) = \frac{1}{SF_{avg}} \tag{A.19}$$

$$F_{fc}(i) = 1 - F_{sc}(i-1) \tag{A.20}$$

$$F_{sc}(i) = 1 - F_{sc}(i-1) + \frac{F_{fc}(i)}{SF_{avg}} \tag{A.21}$$

Simplificando las ecuaciones previas obtenemos lo siguiente:

$$F_{fc}(1) = 1 \tag{A.22}$$

$$F_{fc}(i) = F_{fc}(i-1) - \frac{F_{fc}(i-1)}{SF_{avg}} = F_{fc}(i-1) \cdot (1 - \frac{1}{SF_{avg}}) \tag{A.23}$$

Desplegando la definición recursiva anterior, obtenemos una expresión no recursiva:

$$\begin{aligned}
F_{fc}(i) &= F_{fc}(i-1) \cdot (1 - \frac{1}{SF_{avg}}) \\
&= F_{fc}(i-2) \cdot (1 - \frac{1}{SF_{avg}}) \cdot (1 - \frac{1}{SF_{avg}}) \\
&= F_{fc}(1) \cdot \underbrace{(1 - \frac{1}{SF_{avg}}) \cdot (1 - \frac{1}{SF_{avg}}) \cdots (1 - \frac{1}{SF_{avg}})}_{\text{i-1}} \\
&= 1. \underbrace{(1 - \frac{1}{SF_{avg}}) \cdot (1 - \frac{1}{SF_{avg}}) \cdots (1 - \frac{1}{SF_{avg}})}_{\text{i-1}} \\
&= (1 - \frac{1}{SF_{avg}})^{i-1}
\end{aligned} \tag{A.24}$$

Como mencionamos anteriormente, el rendimiento de la aplicación está determinado por el tiempo que el hilo más lento tarda en ejecutar todas sus instrucciones en una fase paralela. Al emplear BusyFCs, el hilo más lento en una fase paralela dada es asignado a un core rápido en la última iteración del algoritmo A.1. Nótese que puede haber varios hilos ejecutándose en cores rápidos en esta última iteración, pero cualquiera de ellos terminará aproximadamente al mismo tiempo. Como resultado, determinar del tiempo de ejecución de una fase paralela dada se reduce a calcular el número de instrucciones ejecutadas sobre cores rápidos y cores lentos para los hilos asignados a los cores rápidos en la última iteración. Más concretamente, teniendo en cuenta que $NI_{FC,i}$ y $NI_{SC,i}$ denotan el número de instrucciones ejecutadas por el hilo más lento en la fase paralela i-ésima sobre cores rápidos y lentos, respectivamente,

entonces tenemos lo siguiente:

$$NI_{FC,i} = NI_i \cdot F_{fc}(N_{iter}) \tag{A.25}$$

$$NI_{SC,i} = NI_i \cdot (1 - F_{fc}(N_{iter})) \tag{A.26}$$

Donde $N_{iter} = \frac{N-1}{N_{FC}} - 1$. Por último, aproximamos el *speedup* de la aplicación bajo BusyFCs con respecto a una ejecución donde sólo se utilizan cores lentos, como sigue:

$$
\begin{aligned}
Speedup_{BusyFCs} &= \frac{CT_{slow}}{CT_{BusyFCs}} \\
&= \frac{NI \cdot SPIsc}{NI_{FC} \cdot SPI_{fc} + NI_{SC} \cdot SPI_{sc}} \\
&= \frac{\sum_{i=1}^{k}(NI_i \cdot SPI_{sc})}{\sum_{i=1}^{k}(NI_{FC,i} \cdot SPI_{fc} + NI_{SC,i} \cdot SPI_{sc})} \\
&= \frac{\sum_{i=1}^{k}(NI_i \cdot SPI_{fc} \cdot SF_{avg})}{\sum_{i=1}^{k}(NI_{FC,i} \cdot SPI_{fc} + NI_{SC,i} \cdot SPI_{fc} \cdot SF_{avg})} \\
&= \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot SF_{avg})}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_{FC,i} + NI_{SC,i} \cdot SF_{avg})} \\
&= \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot SF_{avg})}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot F_{fc}(N_{iter}) + NI_i \cdot (1 - F_{fc}(N_{iter})) \cdot SF_{avg})} \\
&= \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i) \cdot SF_{avg}}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i \cdot (F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) \cdot SF_{avg}))} \\
&= \frac{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i) \cdot SF_{avg}}{SPI_{fc} \cdot \sum_{i=1}^{k}(NI_i) \cdot (F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) \cdot SF_{avg})} \\
&= \frac{SF_{avg}}{F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) \cdot SF_{avg}} \\
&= \frac{SF_{avg}}{F_{fc}(N_{iter}) \cdot (1 - SF_{avg}) + SF_{avg}} \\
&= \frac{SF_{avg}}{(1 - \frac{1}{SF_{avg}})^{N_{iter}-1} \cdot (1 - SF_{avg}) + SF_{avg}} \\
&= \frac{SF_{avg}}{(1 - \frac{1}{SF_{avg}})^{\frac{N-1}{N_{FC}}} \cdot (1 - SF_{avg}) + SF_{avg}}
\end{aligned}
\tag{A.27}
$$

El objetivo principal de la derivación del *speedup* bajo BusyFCs es ayudar al planificador a realizar asignaciones de hilos a cores. Cualquier cambio en el número de hilos activos de la aplicación (o en su *speedup factor*), obliga al planificador a recalcular el *speedup* nuevamente para reflejar este cambio. Lamentablemente, utilizar la ecuación A.27 repetidas veces en tiempo de ejecución desde el código del SO puede ser demasiado costoso (especialmente por la exponenciación en el denominador). Por lo tanto, esta fórmula no resulta muy apropiada. Sin embargo, es posible aproximarla con una ecuación más sencilla conocida como *utility factor* (UF), que se define como sigue:

$$UF = \frac{SF_{avg} - 1}{(\frac{N-1}{N_{FC}} + 1)^2} + 1 \tag{A.28}$$

El UF permite lograr un equilibrio entre la precisión y el rendimiento en el cálculo. Los detalles sobre la derivación de esta fórmula y su aproximación con la ecuación A.27 se encuentran en [51].

B Detección de fases de SF mediante umbralización

Este apéndice describe el mecanismo empleado para dividir en fases de grano grueso la traza de *speedup factor* obtenida al ejecutar una aplicación secuencial.

Recordemos que la estrategia *Phase-SF* propuesta para obtener un modelo de estimación de SF en tiempo de ejecución para un sistema de AMP (sección 6.3.2) se basa en la detección de las fases de SF. Nuestro objetivo es identificar las distintas fases en una traza de SF. En este contexto, definimos una fase como un conjunto de ventanas de instrucciones contiguas donde el SF permanece constante o no varía significativamente.

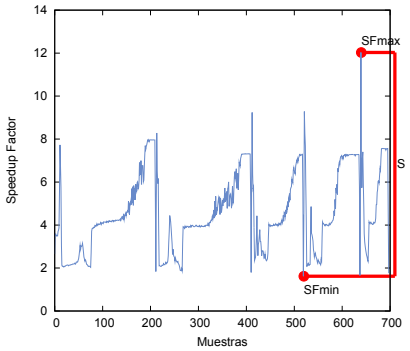


Figura B.1: Mínimo y máximo valor de SF.

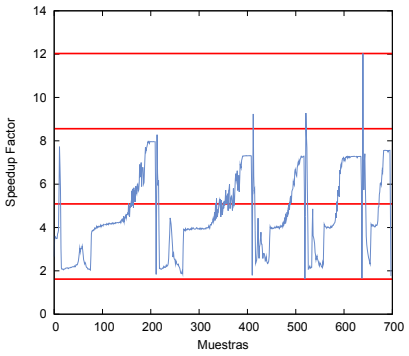


Figura B.2: Rango de SF dividido en subintervalos para U=3.

Varios investigadores han propuesto mecanismos para identificar distintas fases en un programa en tiempo de ejecución empleando soporte hardware [61] [62] o mediante técnicas software, que pueden explotarse desde el planificador del sistema operativo [8]. Por el contrario, nuestro objetivo es detectar estas fases *offline*, a partir de una traza de *SF* obtenida previamente para una aplicación.

Nuestra aproximación para descomponer la traza de *SF* en fases de programa se basa en la utilización de algoritmos de umbralización (*thresholding*) y consta de los siguientes pasos:

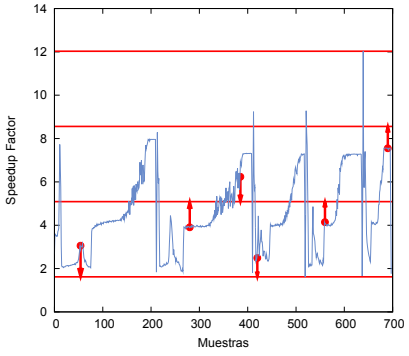


Figura B.3: Aproximación al umbral más cercano.

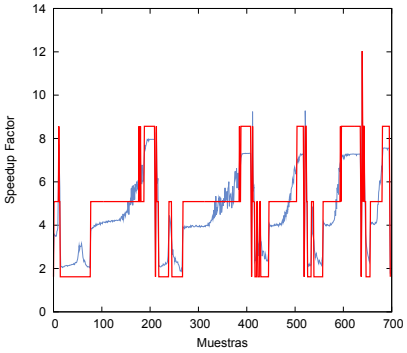


Figura B.4: Fases de SF resultantes.

1. Obtener el máximo y el mínimo valor de SF (SF_{max} y SF_{min}) en la traza de SF. La figura B.1 muestra el máximo y el mínimo valor para la traza de SF del *benchmark mcf* durante la ejecución en el sistema asimétrico QuickIA de Intel. Este *benchmark* pertenece a la suite SPEC CPU.

2. Dividir el rango de SF observado en U subintervalos del mismo tamaño, donde U es un parámetro configurable. Sea p la longitud de cada subintervalo ($p = \frac{S}{U}$) y $S = (SF_{max} - SF_{min})$. En general, cada i-ésimo subintervalo de SF ($i \epsilon \{0..U-1\}$) se define como $[SF_{min} + p.i, SF_{min} + p.(i+1)]$. Por ejemplo, la figura B.2 representa una división potencial en subintervalos para el *benchmark mcf*.

3. Para cada muestra en la traza de SF, obtener el valor de umbral más cercano (límite de un subintervalo). Más específicamente, dado un cierto valor SF, tal que $u \leq SF \leq u + p$, donde u y $u + p$ son los límites de un subintervalo, la función NTV (*Nearest Threshold Value*) que calcula el valor del umbral más cercano se define como sigue:

$$NTV(SF) = \begin{cases} u & si \quad (SF - u) \leq (u + p - SF) \\ \\ u + p & si \quad (SF - u) > (u + p - SF) \end{cases} \tag{B.1}$$

   La figura B.3 muestra como se aplica la función NTV a las muestras de SF tomadas a lo largo del tiempo para el *benchmark mcf*.

4. Finalmente, y utilizando la información obtenida en el paso anterior, dividimos la traza de SF en fases como se muestra en la figura B.4. En particular decimos que dos muestras consecutivas de SF ($SF_j y SF_{j+1}$) pertenecen a la misma fase si $(NTV(SF_j) = NTV(SF_{j+1}))$.

A pesar de su simplicidad, éste esquema nos permite una detección aceptable de fases de SF de grano grueso para los *benchmarks* de las suites SPEC CPU2000 y CPU2006.

El valor más apropiado para el parámetro U es específico de aplicación. Para determinar este valor, empleamos un algoritmo iterativo que encuentra el valor más bajo para este parámetro para el que la desviación media en las fases de SF resultantes alcanzan un cierto umbral. Al utilizar esta técnica, observamos que los valores para el parámetro U comprendidos entre 3 y 5 proporcionan resultados satisfactorios para la mayoría de las aplicaciones consideradas en nuestro estudio.

Es importante destacar, que nuestro objetivo al crear el modelo de estimación de SF es capturar las fases de grano grueso. Por lo tanto, descartamos aquellas fases con muy pocas muestras de SF, que se corresponden con los picos observados en la representación gráfica de la traza.

Bibliografía

[1] Alaa R. Alameldeen y David A. Wood. “IPC Considered Harmful for Multiprocessor Workloads”. En: *IEEE Micro* 26.4 (2006).

[2] A. Annamalai, R. Rodrigues, I. Koren y S. Kundu. “An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs”. En: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. Sep. de 2013, páginas 63-72. DOI: 10.1109/PACT.2013.6618804.

[3] Murali Annavaram, Ed Grochowski y John Shen. “Mitigating Amdahls Law through EPI Throttling”. En: (2005), páginas 298-309.

[4] ARM. *Arm juno development board*. URL: http://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php..

[5] ARM. *Benefits of the big.LITTLE Architecture*. URL: http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf..

[6] ARM. *Coretile express development board*. URL: http://www.arm.com/products/tools/development-boards/versatile-express/coretile-express.php..

[7] Eduard Ayguadé, Bob Blainey, Alejandro Duran, Jesús Labarta, Francisco Martínez, Xavier Martorell y Raúl. Silvera. “Is the schedule clause really necessary in OpenMP?” En: (2003), páginas 147-159.

[8] Michela Becchi y Patrick Crowley. “Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures”. En: (2006), páginas 29-40.

[9] L. Cherkasova, D. Gupta y A. Vahdat. “Comparison of the three CPU schedulers in Xen.” En: (2007), páginas 42-51.

[10] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang y R. Iyer. “QuickIA: Exploring heterogeneous architectures on real prototypes”. En: (2012), páginas 1-8.

[11] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel y Lieven Eeckhout. “Fairness-aware scheduling on single-ISA heterogeneous multi-cores”. En: PACT 13 (2013), páginas 177-187.

[12] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narváez y Joel S. Emer. "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)". En: *Proc. of International Symposium on Computer Architecture, ISCA 12*. 2012, páginas 213-224.

[13] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu y Yale N. Patt. "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems". En: ASPLOS 10 (2010).

[14] Stijn Eyerman y Lieven Eeckhout. "System-Level Performance Metrics for Multiprogram Workloads". En: *IEEE Micro* 28.3 (2008).

[15] Jerome H Friedman. "Stochastic Gradient Boosting". En: *www-stat.stanford.edu/~jhf/ftp/stobst.pdf* (1999).

[16] Ron Gabor, Shlomo Weiss y Avi Mendelson. "Fairness and Throughput in Switch on Event Multithreading". En: (2006).

[17] Matt Gillespie. "Preparing for The Second Stage of Multi-Core HW: Asymmetric (Heterogeneous) Cores". En: *Intel White Paper* (2008).

[18] R. Gonzalez y M. Horowitz. "Energy dissipation in general purpose microprocessors." En: (1996), páginas 1277-1284.

[19] Michael Gschwind. "The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor". En: *Int. J. Parallel Program.* 35.3 (2007), páginas 233-262. ISSN: 0885-7458. DOI: http://dx.doi.org/10.1007/s10766-007-0035-4.

[20] Vishal Gupta y Karsten Schwan. "Brawny vs. Wimpy: Evaluation and Analysis of Modern Workloads on Heterogeneous Processors". En: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, páginas 74-83. ISBN: 978-0-7695-4979-8. DOI: 10.1109/IPDPSW.2013.130. URL: http://dx.doi.org/10.1109/IPDPSW.2013.130.

[21] Mark A. Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann y Ian H. Witten. "The WEKA data mining software: an update". En: *SIGKDD Explor. Newsl.* 11 (1 2009), páginas 10-18. ISSN: 1931-0145.

[22] Hardkernel. *Odroid XU4 board*. URL: http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4.%20Accessed:%202016-6-22..

[23] M. D. Hill y M. R. Marty. "Amdahls Law in the Multicore Era". En: *IEEE Computer* (2008), páginas 33-38.

[24] M. Horowitz, T. Indermaur y R. Gonzalez. "Low-power digital design." En: (1994), páginas 8-11.

[25] Ivan Jibaja, Ting Cao, Stephen M. Blackburn y Kathryn S. McKinley. "Portable Performance on Asymmetric Multicore Processors". En: (2016).

[26] José A. Joao, M. Aater Suleman, Onur Mutlu y Yale N. Patt. "Bottleneck identification and scheduling in multithreaded applications". En: ASPLOS 12 (2012), páginas 223-234.

[27] José A. Joao, M. Aater Suleman, Onur Mutlu y Yale N. Patt. "Utility-based acceleration of multithreaded applications on asymmetric CMPs". En: (2013), páginas 154-165.

[28] David Koufaty, Dheeraj Reddy y Scott Hahn. "Bias Scheduling in Heterogeneous Multi-core Architectures". En: (2010).

[29] Farkas Kumar y Jouppi. "Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction". En: (2003).

[30] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi y Keith I. Farkas. "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance". En: ISCA 04 (2004), páginas 64-.

[31] Viren Kumar y Alexandra Fedorova. "Towards better performance per watt in virtual environments on asymmetric singleISA multi-core systems". En: *SIGOPS Oper. Syst. Rev.* 43.3 (2009), páginas 105-109.

[32] Nagesh B. Lakshminarayana, Jaekyu Lee y Hyesoon Kim. "Age based scheduling for asymmetric multiprocessors". En: (2009), páginas 1-12.

[33] Tong Li y col. "Operating system support for overlapping-ISA heterogeneous multi-core architectures". En: (2010), páginas 1-12.

[34] Tong Li, Dan Baumberger y Scott Hahn. "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin". En: *SIGPLAN Not.* 44.4 (feb. de 2009), páginas 65-74.

[35] Tong Li, Dan Baumberger y David A. Koufaty et al. "Efficient Operating System Scheduling for Performance Asymmetric MultiCore Architectures". En: (2007), páginas 1-11.

[36] Kun Luo, J. Gummaraju y M. Franklin. "Balancing thoughput and fairness in SMT processors". En: (2001), páginas 164-171.

[37] Rasmussen M. *Task placement for heterogeneous mp systems*. 2012. URL: https://lwn.net/Articles/517250/.

[38] Nikola Markovic, Daniel Nemirovsky, Veljko Milutinovic, Osman Unsal, Mateo Valero y Adrian Cristal. "Hardware Round-Robin Scheduler for Single-ISA Asymmetric Multi-core". En: *Euro-Par 2015: Parallel Processing*. Springer, 2015, páginas 122-134.

[39] Nikola Markovic, Daniel Nemirovsky, Osman Unsal, Mateo Valero y Adrian Cristal. "Thread Lock Section-aware Scheduling on Asymmetric Single-ISA Multi Core". En: *IEEE Computer Architecture Letters* 99 (2015).

[40] Sparsh Mittal. "A Survey Of Techniques for Architecting and Managing Asymmetric Multicore Processors". En: *ACM Computing Surveys* (2016).

[41] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan y Vanish Talwar. "Using Asymmetric SingleISA CMPs to Save Energy on Operating Systems". En: *IEEE Micro* 28.3 (2008), páginas 26-41.

[42] Onur Mutlu y Thomas Moscibroda. "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors". En: (2007). DOI: http://dx.doi.org/10.1109/MICRO.2007.40.

[43] J. P. Perez, P. Bellens, R. M. Badia y J. Labarta. “CellSs: Making it Easier to Program The Cell Broadband Engine Processor”. En: *IBM J. Res. Dev.* 51.5 (2007), páginas 593-604. ISSN: 0018-8646. DOI: http://dx.doi.org/10.1147/rd.515.0593.

[44] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars y Lingjia Tang. “Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers”. En: *Proc. of the International Symposium on High Performance Computer Architecture, HPCA15*. 2015, páginas 246-258.

[45] Vinicius Petrucci, Orlando Loques y Daniel Moss. “Lucky Scheduling for Energy-efficient Heterogeneous Multi-core Systems”. En: (2012), páginas 7-7.

[46] Vinicius Petrucci, Orlando Loques, Daniel Mossé, Rami Melhem, Neven Abou Gazala y Sameh Gobriel. “Energy-Efficient Thread Assignment Optimization for Heterogeneous Multicore Systems”. En: *ACM Trans. Embed. Comput. Syst.* 14.1 (ene. de 2015), 15:1-15:26. ISSN: 1539-9087.

[47] Vinicius Petrucci, Orlando Loques, Daniel Mossé, Rami Melhem, Neven Abou Gazala y Sameh Gobriel. “Energy-Efficient Thread Assignment Optimization for Heterogeneous Multicore Systems”. En: *ACM Trans. Embed. Comput. Syst.* 14.1 (2015), 15:1-15:26. ISSN: 1539-9087.

[48] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra y Sanjay Vishin. “Power-performance modeling on asymmetric multicores”. En: (2013), páginas 1-10.

[49] Morten Rasmussen. *Task placement for heterogeneous MP systems*. https://lwn.net/Articles/517250/. Accessed: 2016-07-06. 2012.

[50] Dheeraj Reddy, David Koufaty, Paul Brett y Scott Hahn. “Bridging functional heterogeneity in multicore architectures”. En: (2011).

[51] Juan Carlos Saez. *Planificación de procesos en sistemas multicore asimétricos* URL: http://eprints.ucm.es/12668/1/T32909.pdf.

[52] Juan Carlos Saez, Fernando Castro, Daniel Chaver y Manuel Prieto. “Delivering fairness and priority enforcement on asymmetric multicore systems via OS scheduling”. En: *In Proc. of the International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS13* (2013).

[53] Juan Carlos Saez, Alexandra Fedorova, David Koufaty y Manuel Prieto. “Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems”. En: *ACM TOCS* (2012).

[54] Juan Carlos Saez, Alexandra Fedorova, Manuel Prieto y Hugo Vegas. “Operating System Support for Mitigating Software Scalability Bottlenecks on asymmetric multicore processors”. En: (2010).

[55] Juan Carlos Saez y Jorge Casas Hernán. *PMCTrack - An OS-oriented performance monitoring tool for Linux*. URL: http://http://pmctrack.dacya.ucm.es.

[56] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver y Manuel Prieto-Matías. "ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems". En: (2015).

[57] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver y Manuel Prieto-Matías. "Exploring the Throughput-Fairness Trade-off on Asymmetric Multicore Systems". En: (2014), páginas 326-337.

[58] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova y Sergey Blagodurov. "A Comprehensive Scheduler for Asymmetric Multicore Systems". En: (2010).

[59] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova y Manuel Prieto. "Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems". En: *J. Parallel Distrib. Comput.* 71 (1 ene. de 2011), páginas 114-131. ISSN: 0743-7315.

[60] Daniel Shelepov y col. "HASS a Scheduler for Heterogeneous Multicore Systems". En: *ACM SIGOPS OSR* 43.2 (2009).

[61] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair y Brad Calder. "Discovering and Exploiting Program Phases". En: *IEEE Micro* 23.6 (nov. de 2003), páginas 84-93. ISSN: 0272-1732.

[62] Timothy Sherwood, Suleyman Sair y Brad Calder. "Phase Tracking and Prediction". En: ISCA 03 (2003), páginas 336-349.

[63] Allan Snavely y Dean M. Tullsen. "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor". En: (2000).

[64] Kenzo Van Craeynest y Lieven Eeckhout. "Understanding Fundamental Design Choices in singleISA Heterogeneous Multicore Architectures". En: *ACM Trans. Archit. Code Optim.* 9.4 (ene. de 2013), 32:1-32:23. ISSN: 1544-3566. DOI: 10.1145/2400682.2400691. URL: http://doi.acm.org/10.1145/2400682.2400691.

[65] Y. Zhang, L. Duan, g L. Li B. Pen y S. Sadagopan. "Cross-architecture prediction based scheduling for energy efficient execution on single-isa heterogeneous chip-multiprocessors." En: (2015), páginas 271-285.

[66] S. Zhuravlev, S. Blagodurov y A. Fedorova. "Addressing Cache Contention in Multicore Processors Via Scheduling". En: (2010).

[67] S. Zhuravlev, JC. Saez, S. Blagodurov, A. Fedorova y M. Priteo. "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors". En: (2012).